

Book Collection

Learning Path Complete iOS 12 Development Guide

Become a professional iOS developer by mastering Swift,
Xcode 10, ARKit, and Core ML

Craig Clayton and Donny Wals

Packt
www.packt.com

Complete iOS 12 Development Guide

Become a professional iOS developer by mastering Swift,
Xcode 10, ARKit, and Core ML

**Craig Clayton
Donny Wals**



BIRMINGHAM - MUMBAI

Complete iOS 12 Development Guide

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2019

Production reference: 1200319

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83864-379-9

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry-leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why Subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Craig Clayton is a self-taught, senior iOS engineer at Adept Mobile, who specializes in building mobile experiences for NBA and NFL teams. He also volunteered as the organizer of the Suncoast iOS meetup group in the Tampa/St. Petersburg area for three years and prepared presentations and hands-on talks for this and other groups in the community. His Cocoa Academy online specializes in bringing to the market a diverse list of multi-level iOS courses, such as building apps and games.

Donny Wals is a passionate and curious iOS developer from Netherlands. With several years of experience in building apps and sharing knowledge, Donny is a respected member of the iOS development community. He enjoys delivering talks to share his knowledge and experiences with his peers. In addition to sharing knowledge, he loves learning more about iOS, Apple's frameworks, and development in general. This eagerness to learn has made him into a versatile iOS developer with the knowledge of several Apple frameworks and tools. During WWDC, you will often find Donny binge-watching the talks that Apple engineers deliver to introduce new features and frameworks.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Getting Familiar with Xcode	8
Getting started	9
The Xcode interface	14
Navigator panel	15
Standard editor	15
Utilities panel	15
Debug panel	15
Toolbar	16
Generic iOS device	18
iOS device	18
Connecting wirelessly	23
Window pane controls	25
Summary	27
Chapter 2: Building a Foundation with Swift	28
Playgrounds – an interactive coding environment	29
Data types – where it all starts	32
String	32
Integer data type	33
Floating-point numbers	34
Booleans	34
Variables and constants – where data is held	35
Creating a variable with a string	35
Creating a variable with an integer (int)	35
Debug and print() – detecting your bugs	36
Adding floating-point numbers	36
Creating a Boolean	38
Why constants versus variables?	40
Comments – leaving yourself notes or reminders	40
Type safety and type inference	41
Concatenating strings	41
String interpolation	42
Operations with our integers	43
Increment and decrement	45
Comparison operators	46
Summary	46
Chapter 3: Building on the Swift Foundation	47
Creating a Playground project	48

The if statements – having fun with logic statements	48
Optionals and optional bindings	53
Why optionals?	55
Functions	56
Summary	61
Chapter 4: Digging Deeper	62
Creating a Playground project	63
Ranges	63
Closed range	64
Half-closed range	65
Control flow	66
The for...in loop	67
One-sided range	69
The while loop	71
The repeat...while loop	73
Summary	76
Chapter 5: Digging into Collections	77
Arrays	77
Creating an empty array	78
Creating an array with initial values	79
Creating a mutable array	79
Adding items to an array	80
Checking the number of elements in an array	85
Checking for an empty array	86
Retrieving a value from an array	87
Iterating over an array	91
Removing items from an array	92
Dictionaries	94
Creating a dictionary	95
Adding and updating dictionary elements	97
Accessing an item in a dictionary	100
Iterating over dictionary values	101
Iterating over dictionary keys	102
Iterating over dictionary keys and values	103
Checking the number of items in a dictionary	104
Removing items from a dictionary	106
Sets	108
Creating an empty set	108
Creating a set with an array literal	109
Creating a mutable set	109
Adding items to a set	110
Checking whether a set contains an item	112
Iterating over a set	113
Intersecting two sets	116

Joining two sets	117
Removing items from a set	119
Summary	120
Chapter 6: Starting the UI Setup	121
Useful terms	122
View Controllers	122
Table View Controllers	122
Collection View Controllers	123
Navigation Controllers	124
Tab Bar Controllers	125
Storyboards	126
Segues	127
Stack Views	127
Auto Layout	127
Model View Controller (MVC)	127
App tour	128
The Explore tab	128
Locations	130
Restaurant listings	130
Restaurant detail	132
The Map tab	133
Project setup	134
Creating a new project	134
Summary	137
Chapter 7: Setting Up the Basic Structure	138
Starting from scratch	138
Storyboard setup	140
Adding our app assets	145
Storyboards	157
Creating our launch screen	158
Adding a Navigation Controller	167
Summary	171
Chapter 8: Building Our App Structure in Storyboard	172
Adding a Collection View Controller	175
Hooking up our outlets	177
Creating a custom color	183
Setting up our cell	187
Section header	189
Updating the grid	192
Adding a modal	193
Updating Bar Button Items	200
Unwinding our Cancel button	201
Adding our first Table View	202

Summary	203
Chapter 9: Finishing Up Our App Structure in Storyboard	204
Adding our Restaurant List View	205
Hooking up our outlets	205
Setting up our cell	206
Adding the Reviews View	212
Viewing reviews	214
Map Kit View	216
Summary	221
Chapter 10: Designing Cells	222
Setting up the Explore header	222
Adding Auto Layout to the Explore header	231
Setting up the Explore cell	235
Adding Auto Layout to the Explore cell	236
Setting up the Restaurant cell	237
Adding Auto Layout to the Restaurant cell	243
The Locations cell	247
Summary	248
Chapter 11: Getting Started with the Grid	249
Understanding the Model View Controller architecture	249
Getting familiar with the setup	251
Classes and structures	251
Controllers and classes	258
Understanding Collection Views	259
Creating our controller	266
Understanding Collection View controllers and Collection View cells	269
Getting data into Collection View	272
Understanding the data source	272
Summary	275
Chapter 12: Adding Core Data to Your App	276
Understanding the Core Data Stack	277
Adding Core Data to an existing application	279
Creating a Core Data model	282
Creating the models	283
Defining relationships	285
Using your entities	286
Reading and writing data in a Core Data database	288
Understanding data persistence	288
Persisting your models	289
Refactoring the persistence code	291
Reading data with a simple fetch request	292
Filtering data with predicates	295

Reacting to database changes	296
Implementing NSFetchedResultsController	297
Understanding the use of multiple instances of NSManagedObjectContext	
Summary	303
Questions	304
Further reading	305
Chapter 13: Fetching and Displaying Data from the Network	306
Fetching data from the web	307
Understanding URLSession basics	307
Working with JSON in Swift	308
Updating Core Data objects with fetched data	311
Implementing the fetch logic	314
Updating a movie with a popularity rating	316
Visualizing multiple threads	320
Wrapping the feature up	321
Adding the rating to the movie cell	322
Understanding App Transport Security	323
Observing changes to movie ratings	323
Summary	324
Questions	326
Further reading	326
Chapter 14: Being Proactive with Background Fetch	327
Understanding how background fetch works	328
Looking at background fetch from a distance	329
Looking at background fetch in more depth	329
Implementing the prerequisites for background fetch	331
Adding the background fetch capability	331
Asking iOS to wake your app	332
Updating movies in the background	333
Updating the data model	334
Refactoring the existing code	335
Updating movies in the background	337
Preparing the helper struct	340
Updating the movies	340
Summary	344
Test your knowledge	348
Further reading	348
Chapter 15: Syncing Data with CloudKit	349
Getting familiar with CloudKit	350
Adding CloudKit to your project	351
Exploring the CloudKit dashboard	352
	354

Understanding your CloudKit database	355
Exploring the rest of your container	358
Storing and retrieving data with CloudKit	358
Communicating with CloudKit for the first time	358
Listening for changes in the database	361
Retrieving changes from CloudKit	362
Configuring your AppDelegate	366
Storing data in CloudKit	368
Combining CloudKit and Core Data	370
Preparing the Core Data models for CloudKit	370
Importing CloudKit data	374
Sending Core Data models to CloudKit	376
Summary	379
Questions	380
Further reading	381
Chapter 16: Using Augmented Reality	382
 Understanding ARKit	382
Understanding how ARKit renders content	383
Understanding how ARKit tracks the physical environment	384
 Using ARKit Quicklook	386
Implementing the ARKit Quicklook view controller	386
 Exploring SpriteKit	392
Creating a SpriteKit scene	394
 Exploring SceneKit	395
Creating a basic SceneKit scene	396
 Implementing an Augmented Reality gallery	399
Adding image tracking	399
Preparing images for tracking	400
Building the image-tracking experience	401
Placing your own content in 3D space	405
 Summary	410
 Questions	410
 Further reading	411
Chapter 17: Improving Apps With Location Services	412
 Requesting a user's location	413
Asking for permission to access location data	413
Obtaining a user's location	416
 Subscribing to location changes	418
 Setting up geofences	421
 Summary	423
 Questions	423
 Further reading	424
Chapter 18: Making Smarter Apps with CoreML	425

Understanding machine learning and CoreML	426
Understanding what machine learning is	426
Understanding CoreML	427
Obtaining CoreML models	428
Using a CoreML model	429
Combining CoreML and computer vision	432
Understanding the Vision framework	432
Implementing an image classifier	434
Training your own models with CreateML	437
Training a natural language model	437
Training a Vision model	440
Summary	441
Questions	442
Further reading	443
Chapter 19: Tracking Activity Using HealthKit	444
Understanding HealthKit	445
Requesting access to the HealthKit Store	445
Storing and retrieving data with HealthKit	448
Implementing a workout app	449
Summary	453
Questions	453
Further reading	454
Chapter 20: Streamlining Experiences with Siri	455
Understanding intents	456
Implementing an intents extension	458
Understanding app extensions	459
Configuring your extension	461
Adding vocabularies to your app	463
Adding vocabularies through a .plist file	463
Teaching Siri new vocabularies at runtime	465
Handling intents in your extension	466
Resolving the user's input	467
Confirming the intent status	471
Performing the desired action	472
Adding a custom UI to Siri	476
Implementing Siri Shortcuts	479
Implementing shortcuts through NSUserActivity	480
Donating shortcuts with INteractions	485
Summary	491
Questions	491
Further reading	493
Chapter 21: Using Media in Your App	494
Playing audio and video	495

Creating a simple video player	495
Creating an audio player	497
Implementing basic audio controls	498
Implementing the time scrubber	501
Displaying song metadata	502
Playing media in the background	504
Recording video and taking pictures	507
Taking and storing a picture	508
Recording and storing video	510
Manipulating photos with Core Image	514
Summary	519
Questions	519
Further reading	520
Chapter 22: Implementing Rich Notifications	521
Gaining a deep understanding of notifications	522
Scheduling and handling notifications	526
Registering for notifications	526
Creating notification contents	532
Creating push notifications	532
Creating local notifications	533
Scheduling your notification	534
Scheduling a timed notification	535
Scheduling a calendar-based notification	537
Scheduling a location-based notification	538
Handling notifications	539
Handling notifications in your app	539
Managing pending and delivered notifications	541
Adding actions to notifications	542
Implementing grouped notifications	545
Grouping notifications based on thread identifiers	546
Providing a custom summary message for your notification group	547
Implementing notification extensions	550
Adding a service extension to your app	550
Adding a content extension to your app	553
Summary	560
Questions	560
Further reading	561
Chapter 23: Instant Information with a Today Extension	562
Understanding the anatomy of a Today Extension	562
Finding Today Extensions in iOS	564
Understanding Today Extensions	566
Adding a Today Extension to your app	567
Summary	575
Questions	575

Further reading	576
Chapter 24: Exchanging Data With Drag And Drop	577
Understanding the drag and drop experience	578
Understanding UIDragInteractionDelegate	579
Understanding UIDropInteractionDelegate	580
Implementing a basic drag and drop functionality	582
Adding drag and drop to a plain UIView	583
Adding drag and drop to a UICollectionView	586
Customizing the drag and drop experience	592
Summary	593
Questions	593
Further reading	594
Chapter 25: Improved Discoverability with Spotlight and Universal Links	595
Understanding Spotlight search	595
Adding your app contents to the Spotlight index	597
Indexing your app through user activity	598
Indexing with CSSearchableItem	605
Containing information in CSSearchableItemAttributeSet	605
Adding CSSearchableItem instances to the search index	608
Safely combining indexing methods	611
Handling searchable item selection	612
Understanding Spotlight best practices and ratings	614
Adding metadata to your web links	614
Registering as an indexing delegate	616
Increasing your app's visibility with Universal Links	617
Preparing your server for Universal Links	618
Handling Universal Links in your app	620
Summary	623
Questions	624
Further reading	625
Chapter 26: Extending iMessage	626
Understanding iMessage apps	627
Creating an iMessage sticker pack	628
Optimizing assets for your stickers	629
Creating a custom sticker app	630
Implementing custom, interactive iMessage apps	634
Understanding the iMessage app life cycle	634
Implementing the custom compact view	635
Implementing the expanded view	639
Understanding sessions, messages, and conversations	643
Composing a message	644
Sending a message	645

Summary	646
Questions	647
Further reading	648
Chapter 27: Discovering Bottlenecks with Instruments	649
Exploring the Instruments suite	650
Discovering slow code	654
Closing memory leaks	658
Understanding what a memory leak is	658
Preventing objects from using infinite memory	658
Avoiding reference cycles	660
Discovering memory leaks	662
Creating your own Instrument	667
Adding signpost logging to your app	668
Building an Instruments Package	672
Summary	677
Questions	678
Further reading	679
Chapter 28: Offloading Tasks with Operations and GCD	680
Writing asynchronous code	681
Understanding threads	681
Using dispatch queues in your application	683
Creating reusable tasks with Operations	688
Using Operations in your apps	689
Summary	694
Questions	695
Further reading	696
Chapter 29: Submitting Your App to the App Store	697
Adding your application to App Store Connect	698
Packaging and uploading your app for beta testing	703
Preparing your app for launch	707
Summary	709
Further reading	710
Appendix	711
Other Books You May Enjoy	724
Index	727

Preface

With Apple users spending more money in the App Store, there are plenty of development opportunities for professional iOS developers.

This Learning Path is a direct route to iOS development, which will take you through the basics and help you understand how to put principles into practice. If you are already an experienced programmer, you can gain helpful insights into the latest iOS 12 features with the help of this book. This book is also useful if you're a beginner who wants to gain expertise in iOS development. You'll start with an introduction to iOS development, Xcode, and Swift. You'll also get up to speed with advanced iOS design topics, such as gestures and animations, to give your app the edge. Next, you will understand the latest Swift 4.2 and iOS 12 developments by incorporating new features, such as the latest in notifications, custom-UI notifications, maps, and recent additions in SiriKit. With these tools, you'll not only understand how to program for iOS 12, but also be able to write efficient, readable, and maintainable Swift code that maintains industry best practices. By the end of this book, you will have the confidence to build iOS applications that harness advanced techniques and make best use of the latest features available in iOS 12.

This Learning Path includes content from the following Packt products:

- iOS 12 Programming for Beginners - Third Edition by Craig Clayton
- Mastering iOS 12 Programming - Third Edition by Donny Wals

Who this book is for

If you are completely new to Swift, iOS, or programming and want to become an expert in developing iOS applications, this Learning Path is for you. You'll also find this Learning Path useful if you're an experienced programmer looking to explore the latest iOS 12 features.

What this book covers

Chapter 1, *Getting Familiar with Xcode*, takes you through a tour of Xcode and talks about all the different panels that we will use throughout the book.

Chapter 2, *Building a Foundation with Swift*, deals with the basics of Swift.

Chapter 3, *Building on the Swift Foundation*, teaches us to build on our Swift foundation and learn some further basics of Swift.

Chapter 4, *Digging Deeper*, talks about ranges and control flow.

Chapter 5, *Digging into Collections*, talks about the different types of collections.

Chapter 6, *Starting the UI Setup*, is about building the *Let's Eat* app. We will focus on getting our structure set up using storyboards.

Chapter 7, *Setting Up the Basic Structure*, deals with working on our *Let's Eat* app in a storyboard.

Chapter 8, *Building Our App Structure in Storyboard*, is about adding more to our app structure in a storyboard.

Chapter 9, *Finishing Up Our App Structure in Storyboard*, concludes the discussion of our app structure in a storyboard.

Chapter 10, *Designing Cells*, is about designing the table and collection view cells in a storyboard.

Chapter 11, *Getting Started with the Grid*, concerns working with Collection Views and how we can use them to display a grid of items.

Chapter 12, *Adding Core Data to Your App*, explains that many apps need to store data in a database. This chapter introduces Apple's Core Data framework as a way to include a database for user data in your app.

Chapter 13, *Fetching and Displaying Data from the Network*, shows you how you can access data and resources over a network connection and display the results to the user.

Chapter 14, *Being Proactive with Background Fetch*, explains that iOS allows apps to refresh and load data in the background. You will learn how to implement Background Fetch and you will briefly be introduced to Dispatch Groups.

Chapter 15, *Syncing Data with CloudKit*, covers how to store data in the cloud with CloudKit. It also covers a scenario where CloudKit is used as online storage, while Core Data is used to persist data locally.

Chapter 16, *Using Augmented Reality*, demonstrates how you can use Apple's groundbreaking ARKit framework to build an augmented reality experience.

Chapter 17, *Improving Apps with Location Services*, covers several ways that apps can implement location tracking to enhance and improve a user's experience.

Chapter 18, *Making Smarter Apps with CoreML*, teaches everything you should know about the CoreML framework. Readers will implement a machine learning model that recognizes dominant objects in a scene. This chapter also covers CreateML to augment existing models with more specific training data.

Chapter 19, *Tracking Activity Using HealthKit*, explains how to access a user's health information and how you can implement an app that tracks a user's workouts and stores them in the Health app.

Chapter 20, *Streamlining Experiences with Siri*, shows you how to integrate the SiriKit APIs in your own applications. This enables you to integrate your app deeply into the iOS platform. It also covers the new Siri shortcuts that allow users to quickly perform actions in certain apps.

Chapter 21, *Using Media in Your App*, explains how to play back audio and video, take photos, and apply interesting filters to images that users take with your app.

Chapter 22, *Implementing Rich Notifications*, is a walk-through of everything you need to know about providing a great notification experience for your users. It covers both the UI extension and the content extension.

Chapter 23, *Instant Information with a Today Extension*, shows that developers can add widgets to the Notification Center to disclose quick information to users; this chapter will teach you how.

Chapter 24, *Exchanging Data with Drag and Drop*, takes you through how to augment an app to allow users to drag contents from your app to other apps and vice versa.

Chapter 25, *Improved Discoverability with Spotlight and Universal Links*, explores how you can make an iOS index for your app's contents to make it available through the powerful Spotlight search index.

Chapter 26, *Extending iMessage*, shows how to build a simple sticker pack and app for iMessage.

Chapter 27, *Discovering Bottlenecks with Instruments*, explains how to profile your app's performance with instruments. You also learn how to implement and use your own instrument tools.

Chapter 28, *Offloading Tasks with Operations and GCD*, covers apps that perform increasingly complex tasks. You will learn how to ensure that complex or slow tasks don't freeze your user interface.

Chapter 29, *Submitting Your App to the App Store*, explains how to distribute your app to beta testers through TestFlight and how to submit your app for review in order to publish it to the App Store.

To get the most out of this book

All sample code in this book was written in Swift 4.2 with Xcode 10.0 on a Mac running macOS Mojave. To follow along with all the examples in this book you must have at least Xcode 10.0 installed on your machine. It is recommended that you also have at least macOS Mojave installed on your Mac because not all code samples are compatible with older versions of macOS.

This book assumes that you are somewhat familiar with Swift and iOS development. If you have no experience with Swift at all, it is recommended that you skim through Apple's Swift manual and go over the basics of iOS development. You don't have to be an expert on iOS development yet, but a solid foundation won't hurt you since the pacing of the book is aimed at somewhat experienced developers.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Ziipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Complete-iOS-12-Development-Guide>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded WebStorm-10*.dmg disk image file as another disk in your system."

A block of code is set as follows:

```
func doSomething(completionHandler: (Int) -> Void) {  
    // perform some actions  
    var result = theResultOfSomeAction  
    completionHandler(result)  
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

The answers to the questions posed in this book can be found in the appendix.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Getting Familiar with Xcode

So, you want to get into iOS development? I was in your shoes on January 27, 2010, when Apple first announced the iPad. As soon as the conference was over, I knew that I wanted to learn how to create apps for the iPad. I signed up for the Apple Developer website and paid my \$99 annual fee. But then, I realized that I did not know where to begin. A large variety of instructional books or videos did not exist, especially since the iPad hadn't released. I had previous programming experience; however, I had no idea how to write Objective-C (the original programming language for iOS). Therefore, I had to teach myself the basics. In this book, we will learn what it takes to become an iOS developer together.

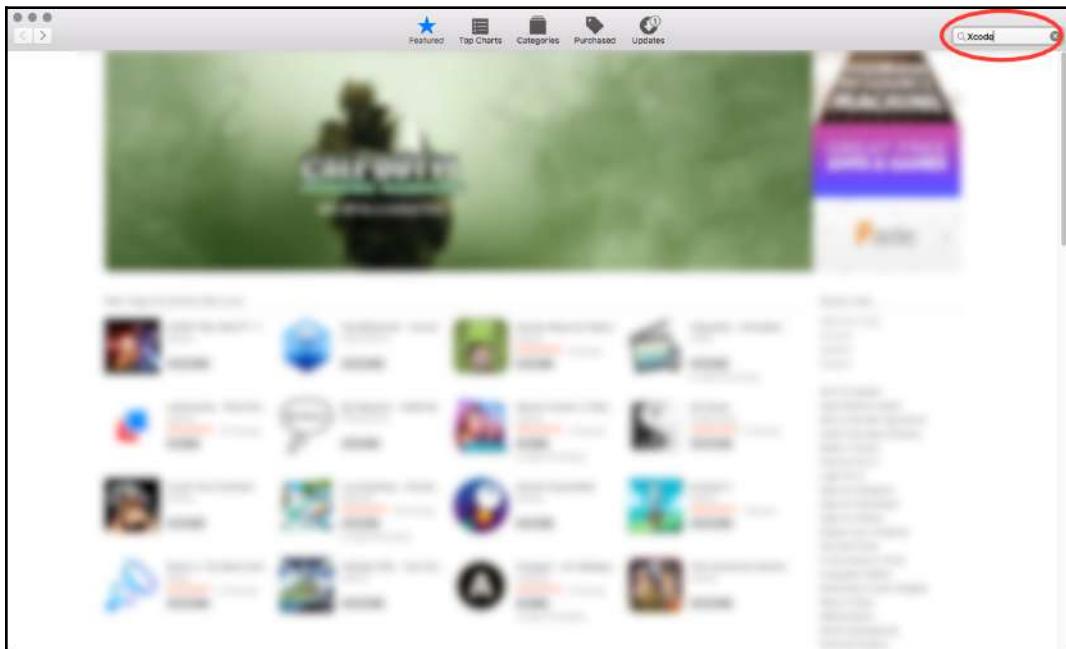
If you are new to programming, take your time. You should understand the lessons that are provided in one chapter before moving on to the next. These essential skills will set you up with a solid foundation in iOS development. If you have previous programming experience, you should still review the earlier chapters, as they will be a refresher for you.

Throughout this book, we will work in Xcode, specifically Xcode 10 (and Swift 4, which we will tackle later in this book). Xcode is known as an **Integrated Development Environment (IDE)**. Using Xcode gives us everything we will need to build apps for iOS, tvOS, macOS (formerly, OS X), and watchOS. In this chapter, we will explore Xcode to help you get more comfortable using it. If you are not on Xcode 10, make sure to update Xcode, as the code in this book will not run correctly otherwise.

Our focus in this book will be on creating a universal iOS app (an app for both the iPhone and iPad). The best way to do this is to create a project to familiarize yourself with where everything is and how to find what you need. So first, let's first download and install Xcode.

Getting started

To download Xcode, launch the App Store on your Mac and then type `Xcode` into the search bar in the upper-right corner:

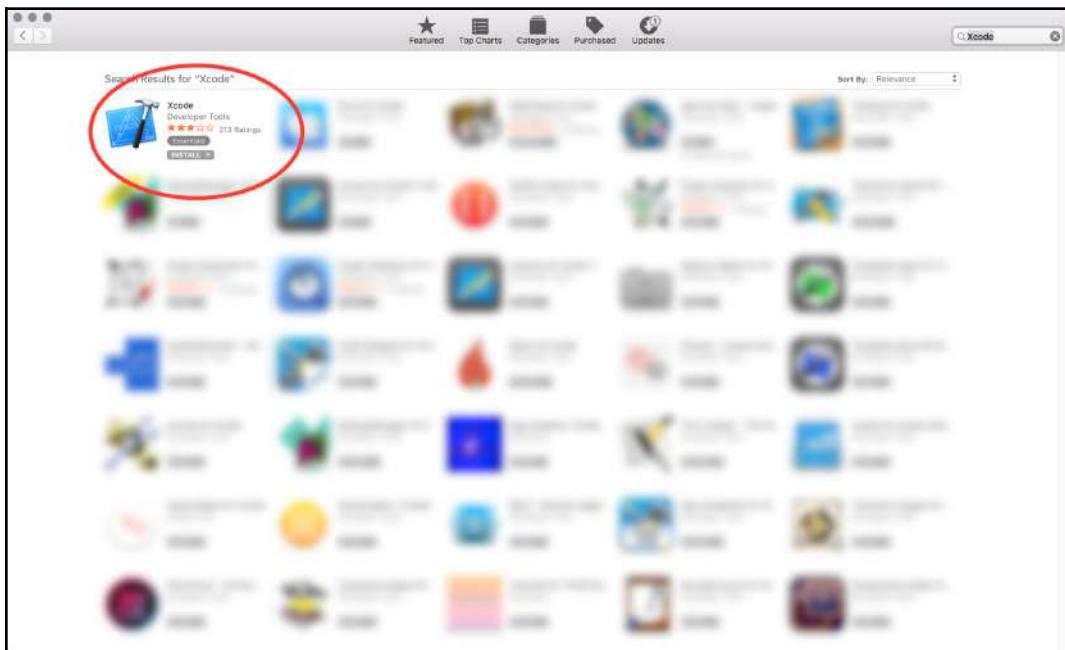


This is the screenshot of the App Store and the blurred out information is not important here



For enhanced image quality, download the graphics bundle from https://www.packtpub.com/sites/default/files/downloads/9781789348668_ColorImages.pdf.

Next, click on **INSTALL**:

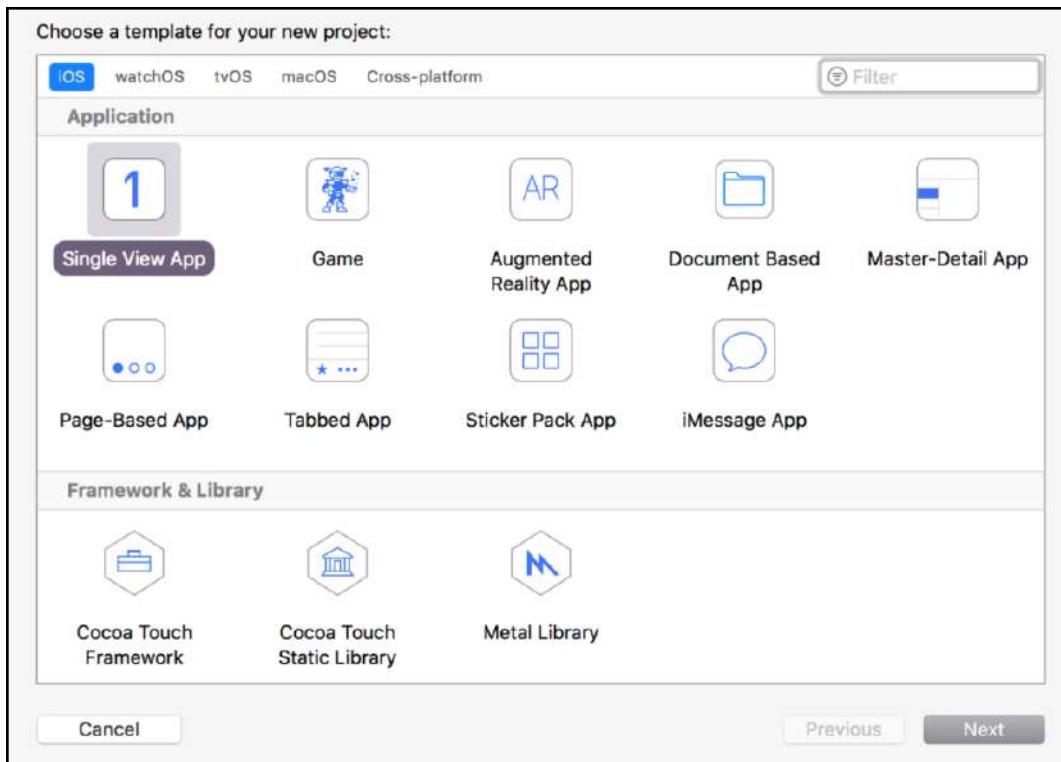


This is the screenshot of the App Store and the blurred out information is not important here

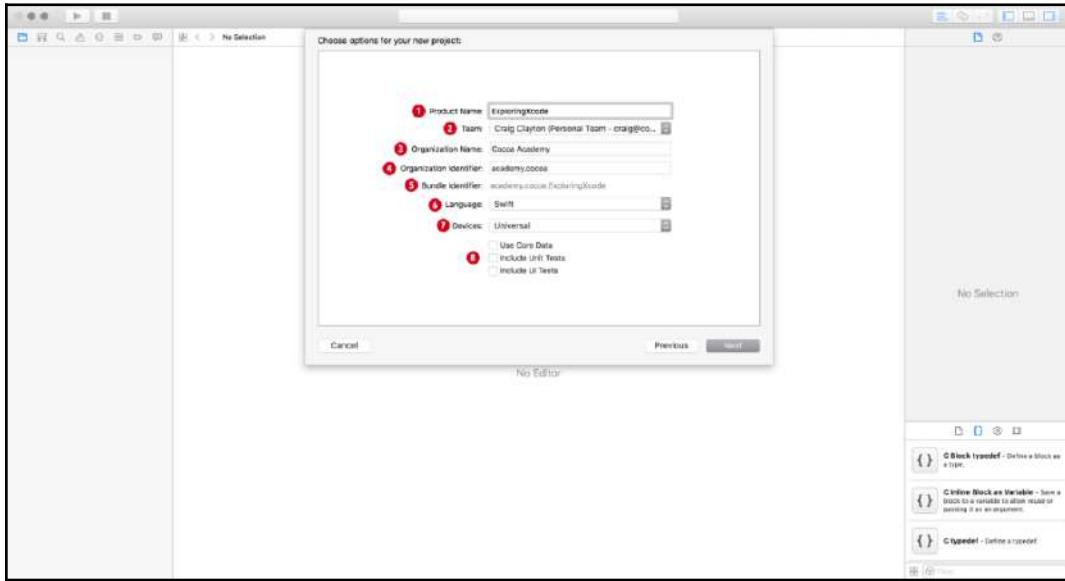
Once installed, launch Xcode, and you should see the following **Welcome to Xcode** screen:



If this is the first time you have launched Xcode, then you will see **No Recent Projects** in the right-hand panel. If you have previously created projects, then you will see those listed to the right. To get started, we are going to click on **Create a new Xcode project** in the left-hand panel of the welcome screen. You will see the new project screen, as follows:



Across the top of this screen, you can select one of the following items: **iOS**, **watchOS**, **tvOS**, **macOS**, and **Cross-platform**. Since we are creating apps for iOS, make sure that you have iOS selected. Then, choose **Single View App** and click on **Next**. Now, you will see an options screen for a new project:



This option screen has the following seven items to complete or choose:

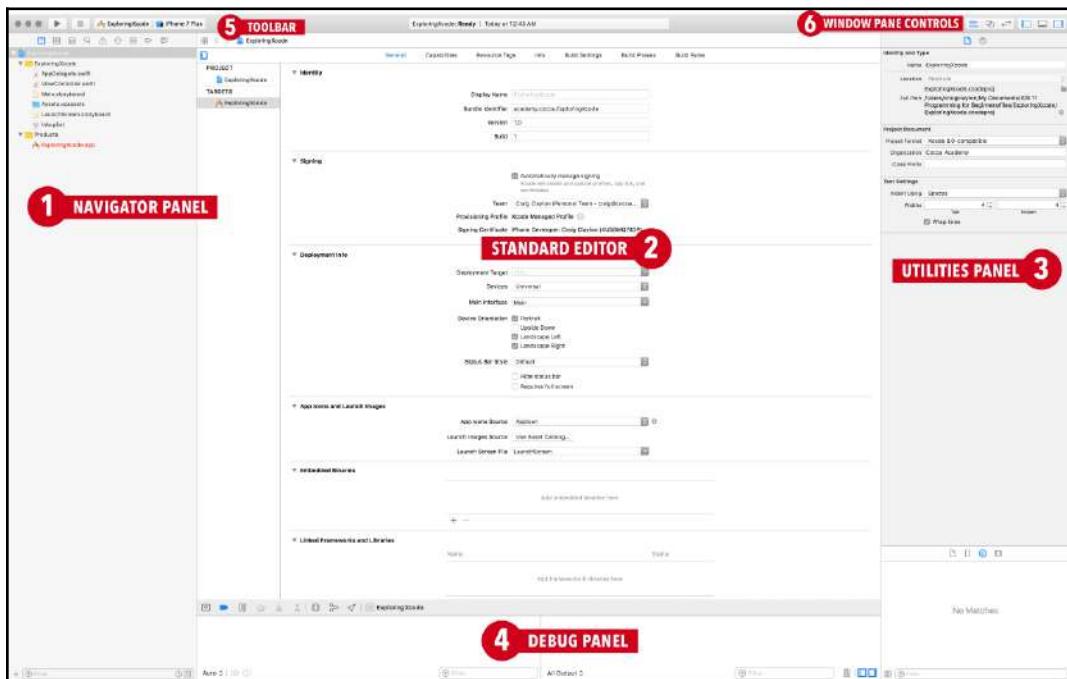
1. **Product Name:** The product name is your app. We are going to set ours as ExploringXcode.
2. **Team:** The team connects to your Apple account. We are going to ignore this for now, because we do not need the Team for this chapter. If you already have a team set up, leave it as is. We will cover this in greater detail later in this book.
3. **Organization Name:** You can set the organization name to your company name, or just use your name.
4. **Organizer Identifier:** You will set the organizer identifier to be your domain name in reverse. For example, my website URL is `cocoa.academy`, and therefore, my identifier is `academy.cocoa`. Since URLs are unique, it will ensure that no one else will have your identifier. If you do not have a domain, then use your first and last names for now. However, you will eventually have to purchase a domain if you want to submit your app to the Apple Store.

5. **Bundle Identifier:** When you create a new project, Apple will combine your **Product Name** with your **Organizer Identifier** to create your unique bundle identifier. So, even if 10,000 people create this project, each person will have a different bundle identifier.
6. **Language:** Set language to **Swift**.
7. **Checkboxes:** You can uncheck **Use Core Data**, **Include Unit Tests**, and **Include UI Tests**, as these are things that we will not use in this chapter.

Now, select **Next**, and Xcode will prompt us to save our project. I have a dedicated folder for all my projects, but you can save it on your desktop for easy access.

The Xcode interface

Your project is now open, and it is time for us to get familiar with all of the panels. If this is your first time in Xcode, then it will probably be a bit overwhelming for you. Therefore, we will break it down into six parts:



- NAVIGATOR PANEL
- STANDARD EDITOR
- UTILITIES PANEL
- DEBUG PANEL
- TOOLBAR
- WINDOW PANE CONTROLS

Navigator panel

The primary use of the navigator panel is to add new files and select existing files. The other icons are used from time to time; we will cover them as we need them.

Standard editor

The standard editor is a single panel view that's used to edit files. The standard editor area is the primary area in which you will work. In this area, we can view storyboard files, see our Swift files, or view our project settings.

Utilities panel

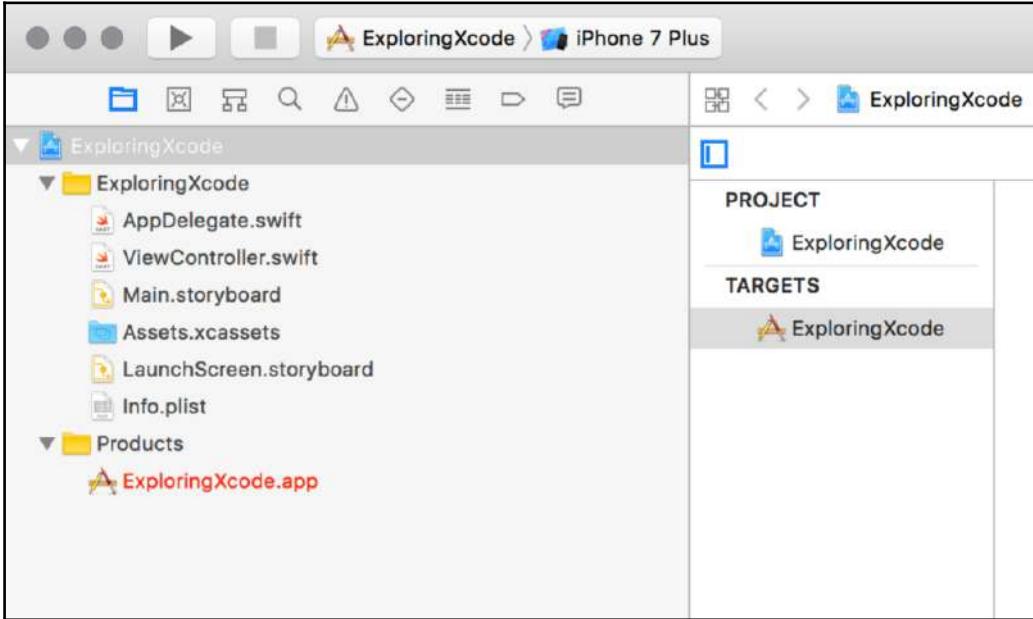
The utilities panel can be a bit confusing when you first use Xcode because this menu changes based on what you have selected in the standard editor. When we start building an app, we will dig deeper into this. For now, know that the utilities panel is made up of the inspector pane at the top and the library pane at the bottom. The inspector pane allows you to change the attributes or properties of things you put in your storyboard; the library pane enables you to insert objects, image assets, and code snippets into your app.

Debug panel

The debug panel will allow us to see log messages from our app. You will become very familiar with this panel by the time you finish this book. The debug panel is one of the most excellent tools for getting feedback on what your app is doing or not doing.

Toolbar

Next, we look at the toolbar, which is demonstrated as follows:



First, we have a play button, which is how we launch our app (or use *command + R*). Next, you will see a stop button, which will not be active until you run your app. This stop button (or *command + .*) is used to stop your app from running. To the right of the stop button, you will see your target (your project name), along with the current simulator that has been selected.

If you click on your project name, you will see a screen similar to this:



This drop-down menu, which we will call the **Device** and **iOS Simulators** drop-down menu, allows you to change your simulator type. For our project, select **iPhone7 Plus** as your simulator and then click on the play icon (or use *command + R*) to run your app.

Now, let's return to Xcode and select the stop button (or use *command + .*).

If you use the keyboard shortcut, make sure Xcode is in focus; otherwise, this shortcut will not work. I work on a 15-inch MacBook Pro Retina. Therefore, when I am working on an app, I will use the iPhone X or iPad Air 2 simulator in landscape mode. They both fit nicely on my screen without me having to resize either.

In addition to the **Simulator**, there is a **Build Only Device** as well as a **Device** section, both of which can be found at the top of the **Device** and **Simulator** drop-down menu that was shown earlier in this chapter. Note that, for our purposes, you will only need a simulator while we are building the app; however, you can add an iOS device if you would like (see under **iOS Device**).

Generic iOS device

The **Generic iOS Device**, under the **Build Only Device** section of the **Device** and **Simulator** drop-down menu, is used for when you need to archive your app, which means that you are preparing your app for submission to Apple (either to the App Store or Test Flight). If you try to select **Generic iOS Device** now and run the app, you will get the following message:



Therefore, change **Generic iOS Device** to an actual simulator, and then you will be able to continue.

iOS device

If you do not have a device connected to the computer, you will see **No devices connected** under the **Device** section of the **Device** and **Simulator** drop-down menu.

As noted earlier, when we start building the *Let's Eat* app, you will have the option of using the simulator or connecting a device to Xcode. Using a device is slower; however, the simulator will not perform in the same way as a device will.

In the past, you needed to have a paid account to build your app on a device. Nowadays, you do not need a developer account to run the app on your device. Note that, if you decide to connect your device instead of using a simulator, you will need iOS 12 installed on it. Xcode 10 introduced the capability of connecting your phone wirelessly. We will look at the traditional way first and then we will go over how you can connect your phone wirelessly.

The following steps are only intended for those of you who do not want to pay for the Apple Developer Program at this time:

1. Connect your iOS device via USB.
2. In the drop-down menu, select your device (here, I have chosen **Xclusive iPhone 6 Plus**):



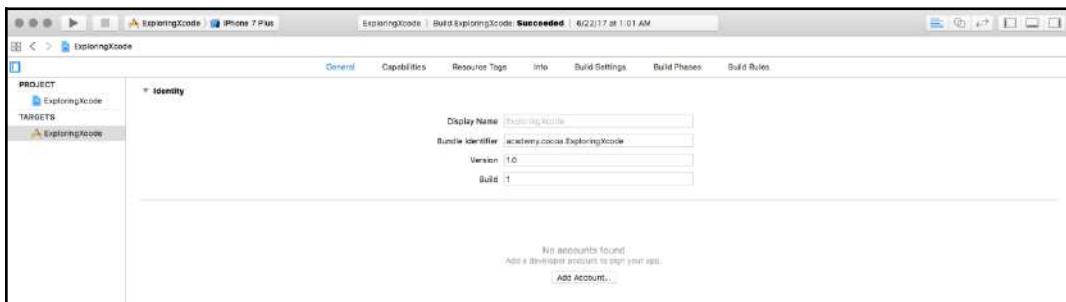
3. Wait for Xcode 10 to finish indexing and processing. The indexing and processing may take a bit of time. Once complete, the status will say **Ready**.
4. Run the project by hitting the Play button (or use *command + R*).

You will get two errors that state the following:

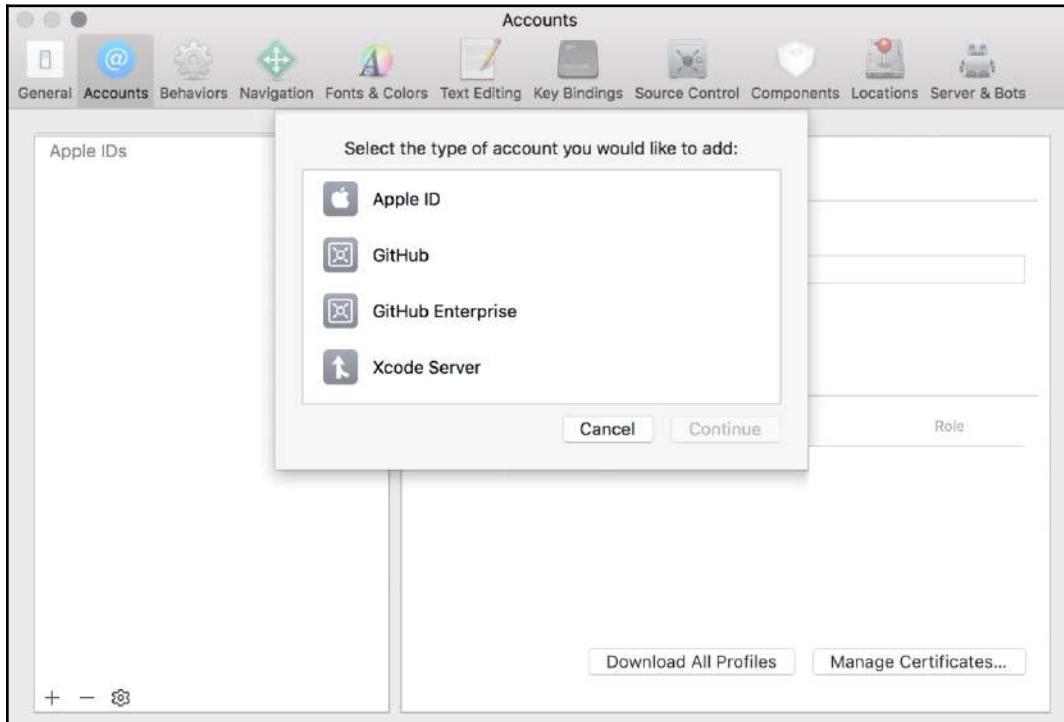
- Signing for ExploringXcode requires a development team. Select a development team in the project editor.
- Code signing requires a product type application in SDK iOS 12.0.

Ignore the specifics of these errors as they indicate that we need to create an account and add our device to that account.

5. Now, in the standard editor, you will see under **Signing** that you need to add an account:

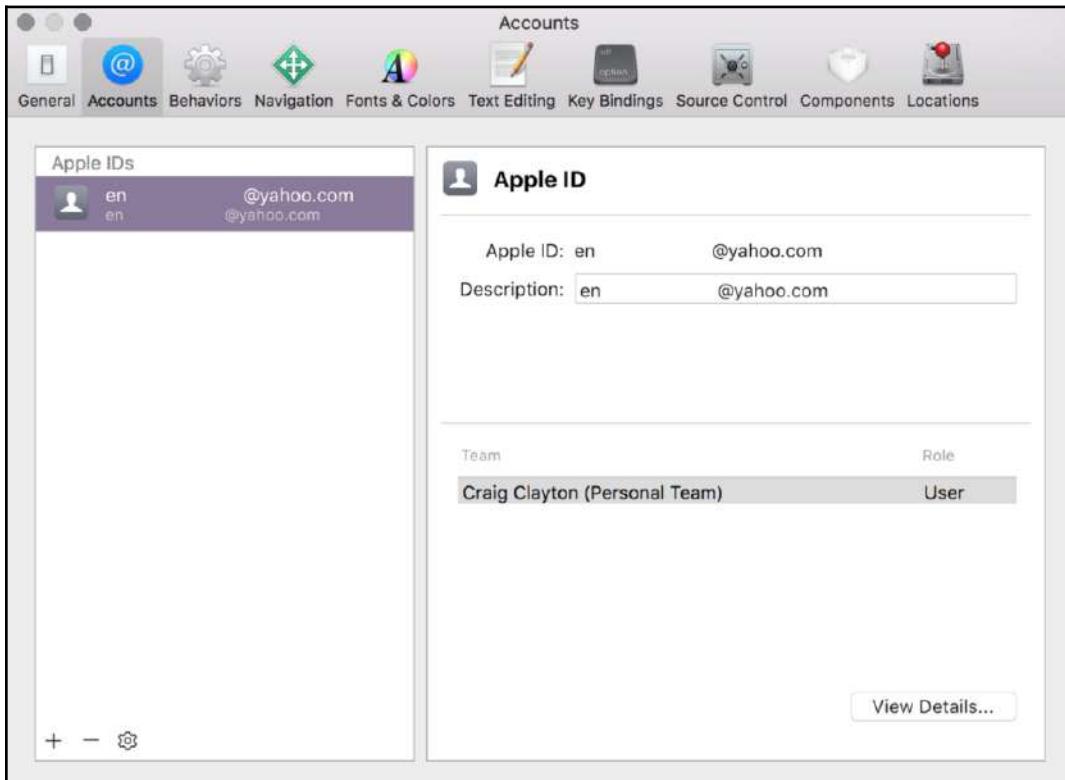


6. Click on **Add Account**. If a **Sign into Xcode with your Apple ID** dialog box does not pop up, inside the **Accounts** screen on the bottom left, click on the **+** and select **Apple ID**:



7. Then, when you click on **Create Apple ID**, you will be asked to enter your birth date, name, email, and password, along with a number of security questions. Make sure that you verify your email before you answer the security questions, otherwise you will have to come back to this screen and add your **Apple ID** again.

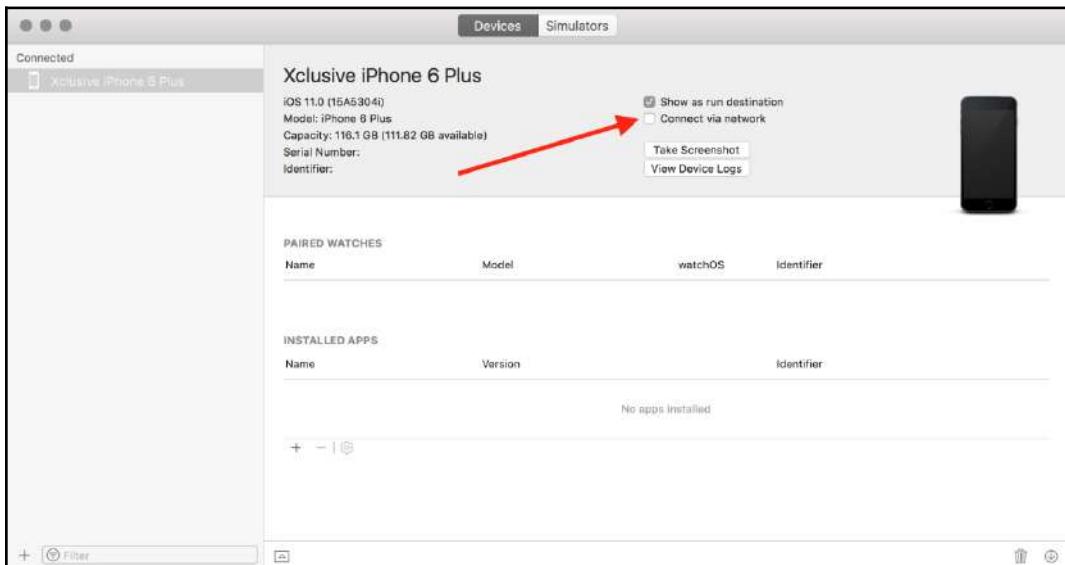
8. Once you have finished all of the steps, you will see your account, as follows:



If you already have an account, then instead of seeing **Add Account**, you will see a drop-down menu with your account listed. If your device is not connected to this account, you might see a message asking if you would like to add your device to your account. Adding your device to an account is for testing purposes only.

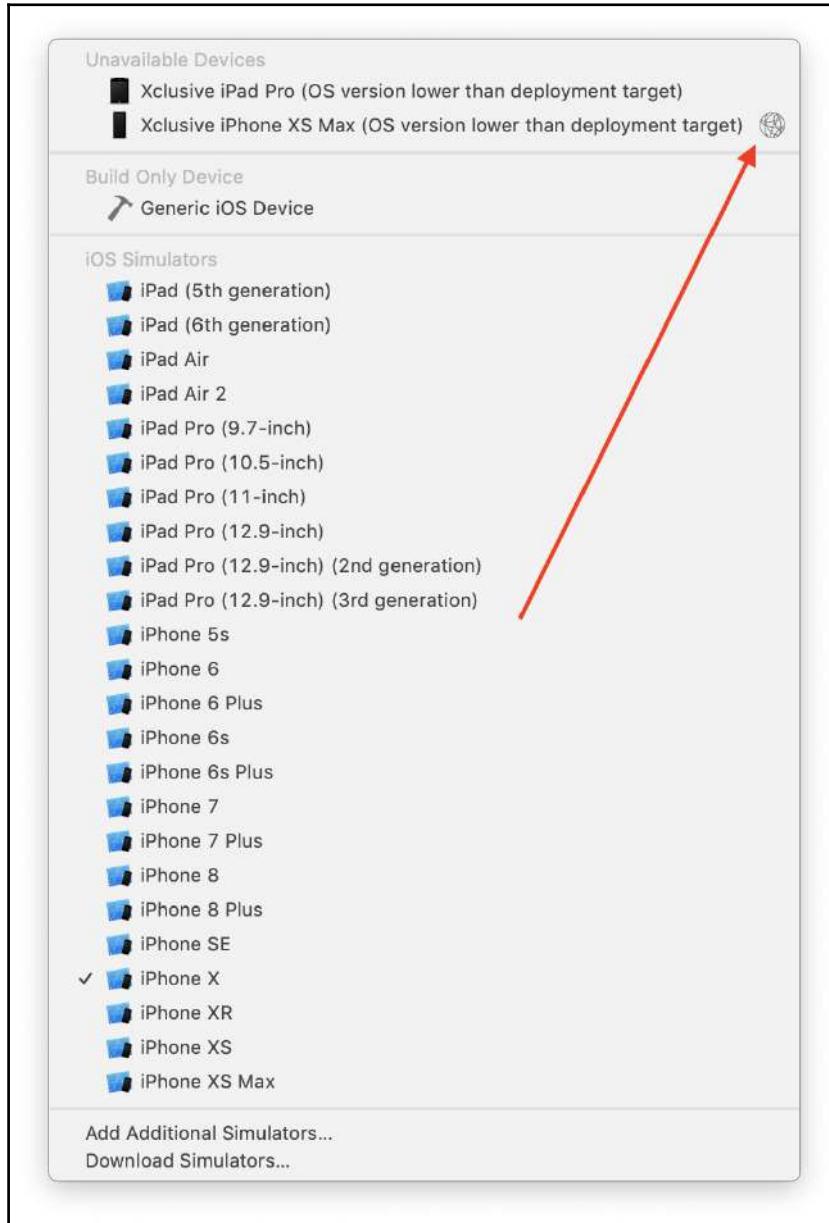
Connecting wirelessly

Now that you have your phone and account connected, you can quickly get your phone set up to run wirelessly. With your device already connected via USB, go to **Window | Devices**, and then **Simulators**. Click on the checkbox marked **Connect via network**:



Make sure that your phone and your computer are connected to the same Wi-Fi network.

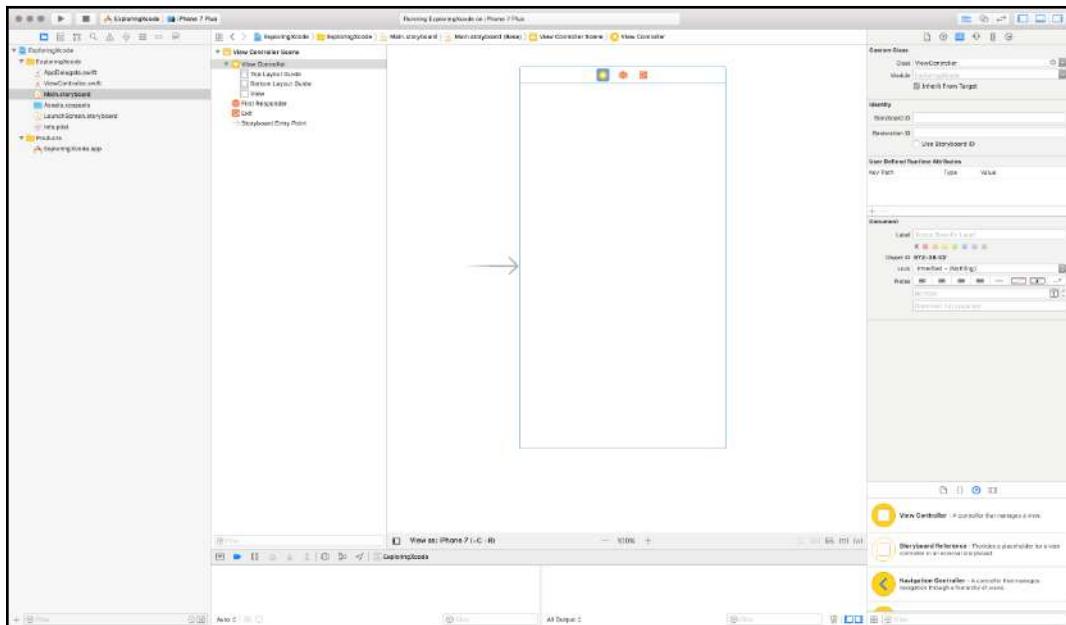
When I first connected to my device, I saw a globe icon in Xcode that lets you know that your device is connected via the network, as demonstrated in the following screenshot:



After a short time, the globe went away. Even if you do not see the icon, you can disconnect the USB, and your device should still be connected to Xcode (as long as it is connected to the same Wi-Fi network).

You will not need to use a device for this book, but it is always good to run your app in an actual device before you submit it to the store.

Before we get to the right-hand side of the toolbar, select the `Main.storyboard` file in your navigator panel. This file is used to display all of your visual setup for your entire app. We will this in detail later in this book. After you select the file, you should see the following:

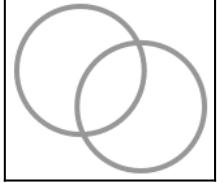


Window pane controls

The following screenshot shows the window pane controls:

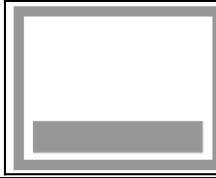


Moving on to the window pane controls, you will see two groups of icons. The first group is called the Editor Mode, and the second group is called the View. Let's look at the functions of the Editor Mode icons:

Editor Mode icons	Function
	This icon controls the standard editor (which is the center panel in the earlier screenshot of the <code>Main.storyboard</code> file in the navigator panel).
	This icon splits the Standard editor into two panels, where you will see the <code>ViewController.swift</code> file on the right. We will use this split screen throughout this book.
	This icon is the Version editor. We will not address the Version editor in this book since it is a more advanced feature.

At this point, you might be thinking that there are way too many panels open, and I would agree with you! The last panel is where the previous group of View icons in the toolbar comes in handy.

Let's look at these icons and their functions in the following table:

View Mode icons	Function
	This icon will toggle (hide or show) the navigator panel (or use <i>command + O</i>).
	This icon will toggle (hide or show) the debug panel (or use <i>command + shift + Y</i>).
	This icon will toggle (hide or show) the utilities panel (or use <i>command + alt + O</i>).

Summary

Congratulations! You have finished exploring the basics of Xcode. When we start building our app, we will cover the more essential parts of Xcode in depth. In the next few chapters, we will begin learning about the Swift programming language. We will use the latest Swift version, and this will be a basic intro into the programming language. If you are familiar with Swift, feel free to skip ahead to [Chapter 6, Starting the UI Setup](#). Even if you are familiar with Swift, it is always good to go back through the basics as a refresher. So, let's get started!

2

Building a Foundation with Swift

Now that we have had a short tour of Xcode, it is time to start learning about Swift. Remember, if you are new to programming, things will be very different for you, so take your time. The essential skills that you learn here will set you up with a solid foundation in iOS development. If you have previous programming experience, you should still review this chapter, as it can only enhance your programming skills and act as a refresher for you.

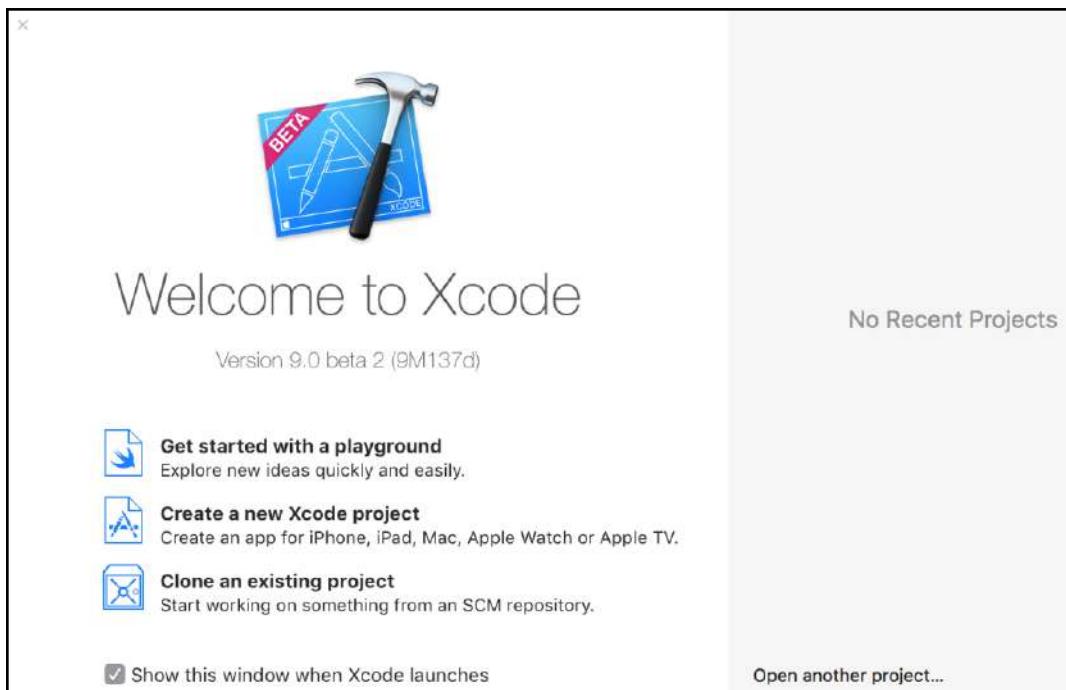
On June 2, 2014, Apple changed the game for iOS development, because this was the day they announced Swift to the world. With this announcement, everybody was put on an even playing field, because they had to learn a new programming language. Swift has brought a more modern approach to developing apps and has seen a massive influx of new developers of all ages wanting to build iOS apps. However, enough about history! Let's dig in and look at what you are going to learn about.

The following topics will be covered in this chapter:

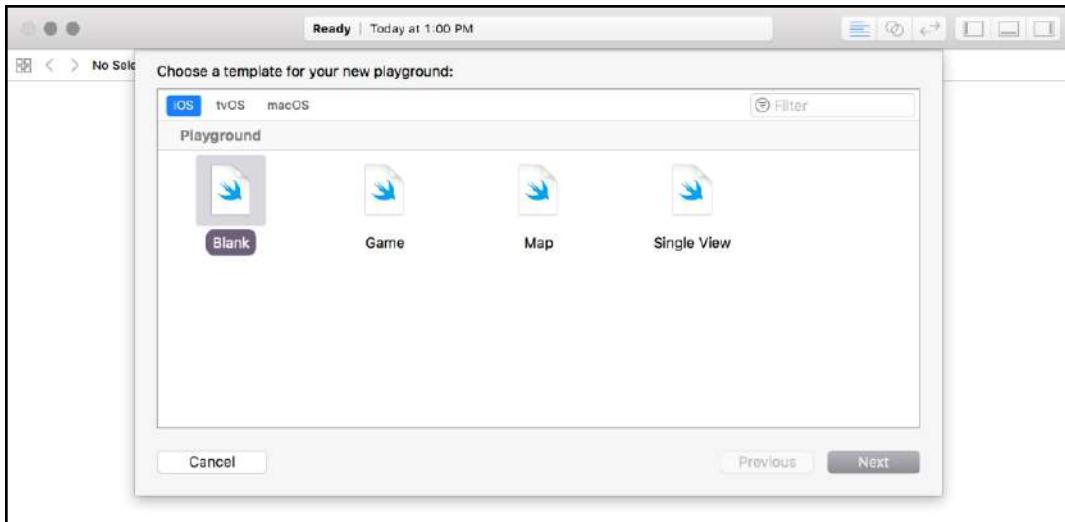
- Playgrounds
- Data types
- Variables and constants
- Debug and `print()`
- Comments

Playgrounds – an interactive coding environment

Before we jump into building the app that we will be creating in later chapters, called *Let's Eat*, we need to understand the basics of Swift. An easy way to experiment with Swift is to use **Playgrounds**. It is an interactive coding environment that evaluates your code and displays the results. Using Playgrounds gives us the ability to work with Swift without needing to create a project. It is great for prototyping a particular part of your app. So, whether you are learning or experimenting, Playgrounds are an invaluable tool. To create a Playground, we need to launch Xcode and click on **Get started with a playground**:

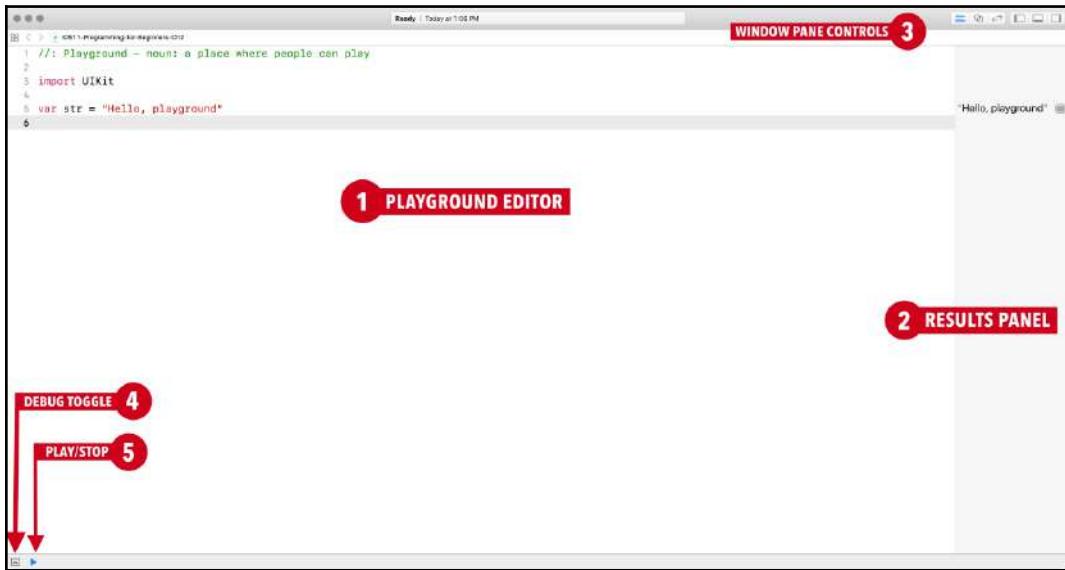


The **Playground** template screen appears. Make sure that you select **iOS**, and then choose **Blank** and hit **Next**:



You will be asked to give your project a name and a location to save the file; name your new project `Playground_iOS11-Programming-for-Beginners-Ch2`. You can save the file anywhere that you like. Now, with the project saved, we can explore Playgrounds in a little more detail.

When you launch the app, you will see five distinct areas:



Let's break down each area in Playgrounds:

- **Playground Editor:** This area is where you write all of your code.
- **Results Panel:** The Results panel is a feature that's only found in Playgrounds and provides immediate feedback.
- **Window Pane Controls:** The Window Pane Controls have two groups of icons:



As we discussed earlier, the first group is called the **Editor Mode**, and the second group is called the **View**. Refer to the detailed description of these icons in the previous chapter for information about what each one does.

- **Debug Toggle:** This button allows you to hide and show the Debug panel and toggle on the Debug panel.
- **Play/Stop:** This button is used to make Playgrounds execute code or to stop Playgrounds from running. Typically, Playgrounds runs on its own, but sometimes you need to manually toggle this feature on when Playgrounds does not execute your code for you.

Now that we have the Xcode panels setup, delete all of the code in this file. Your Playground should have three open panels: your Playground Editor, the Results Panel, and the Debug Panel. Let's start digging into some code.

Data types – where it all starts

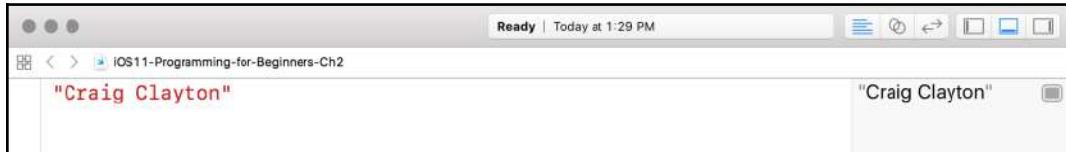
Swift offers a collection of built-in data types. Its data types are a string, an integer, floating-point numbers, and Booleans. These data types are found in most programming languages. Therefore, if you are not new to programming, you can skip this section and start at the *Variables and constants – where data is held* section later.

Let's walk through each data type for those of you who are new to programming or would like a refresher.

String

The first data type we will cover is a string. A series of characters represent a string. Strings are used to display text in an app. A string wrapped in quotes is known as a string literal. In programming, we cannot just add text to Playgrounds. So, to write a string, we must wrap our string inside quotes.

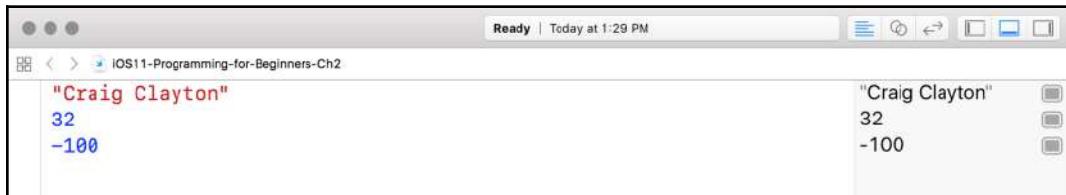
Let's add our name into Playgrounds, wrapped in quotes:



In Playgrounds, your values appear inside of your Results Panel. So, we now know that in order to create a string, we need to use quotes.

Integer data type

Integers (Ints) are whole numbers, such as 32 and -100. Integers are useful when you need to perform calculations (that is, adding, subtracting, multiplication, and so on). Let's add some numbers to Playgrounds. On the next line, under your name, type 32, and then, on the following line, -100, as demonstrated in the following screenshot:



Again, you see both 32 and -100 in the Results Panel under your name.

Floating-point numbers

Floating-point numbers are numbers with a fractional component, such as 4.993, 0.5, and -234.99. Let's add these values to Playgrounds as well:

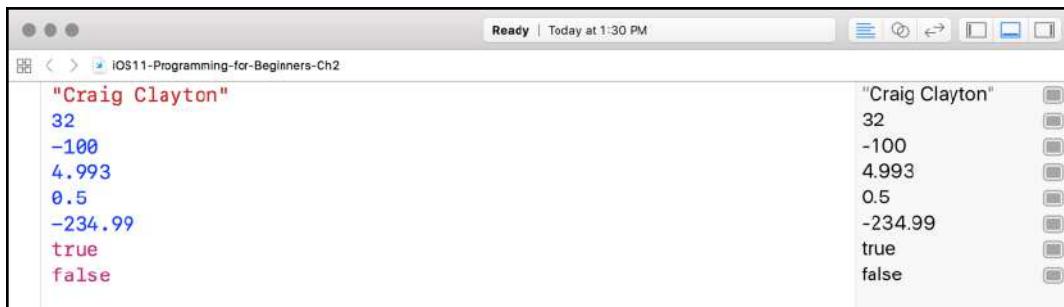


The screenshot shows the Xcode Playgrounds interface with the title "iOS11-Programming-for-Beginners-Ch2". On the left, a list of variables is shown with their values in pink text. On the right, the same variables are listed again with their values in blue text. The variables and their values are:

Variable	Value (Pink)	Value (Blue)
"Craig Clayton"	"Craig Clayton"	"Craig Clayton"
32	32	32
-100	-100	-100
4.993	4.993	4.993
0.5	0.5	0.5
-234.99	-234.99	-234.99

Booleans

Booleans (**bools**) are referred to as logical because they can either be true or false. Use Booleans when you need to determine whether some logic is `true` or `false`. For example, did the user log in? This statement would either be true—yes they did, or false—no they did not. So, in Playgrounds, add `true` and `false`:



The screenshot shows the Xcode Playgrounds interface with the title "iOS11-Programming-for-Beginners-Ch2". On the left, a list of variables is shown with their values in pink text. On the right, the same variables are listed again with their values in blue text. The variables and their values are:

Variable	Value (Pink)	Value (Blue)
"Craig Clayton"	"Craig Clayton"	"Craig Clayton"
32	32	32
-100	-100	-100
4.993	4.993	4.993
0.5	0.5	0.5
-234.99	-234.99	-234.99
true	true	true
false	false	false

Now, we have covered all of the primary data types in Swift. Right now, we have no way to use these data types. Using the data is where variables and constants come into play.

Variables and constants – where data is held

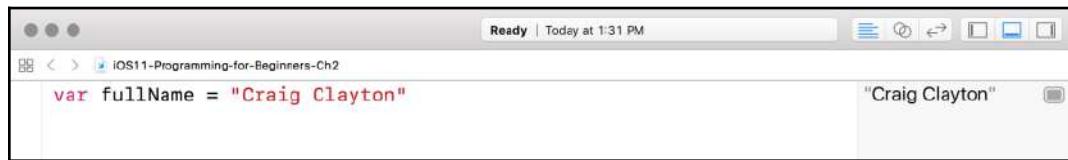
Variables and constants are like containers that contain any data. When you want to declare a variable, you have to use the var keyword. Let's declare each of the data types we did earlier, but, this time, using variables and constants instead.

Creating a variable with a string

First, delete what you have entered in Playgrounds already, and let's declare our first variable, named `fullName`, and set it to your name:

```
var fullName = "Craig Clayton"
```

The preceding code says that we have a variable named `fullName` and that it is holding a string value of `Craig Clayton`. Your Results Panel shows your actual name as its data:

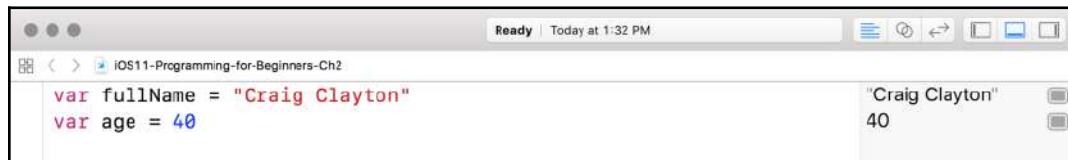


Creating a variable with an integer (int)

Now, let's create a variable with an `int` called `age` and set it to our age (or whatever you want your age to be) by adding the following:

```
var age = 40
```

Our program now knows that `age` is an `int`. You should see both your name and `age` in the Results Panel, just like you did previously:

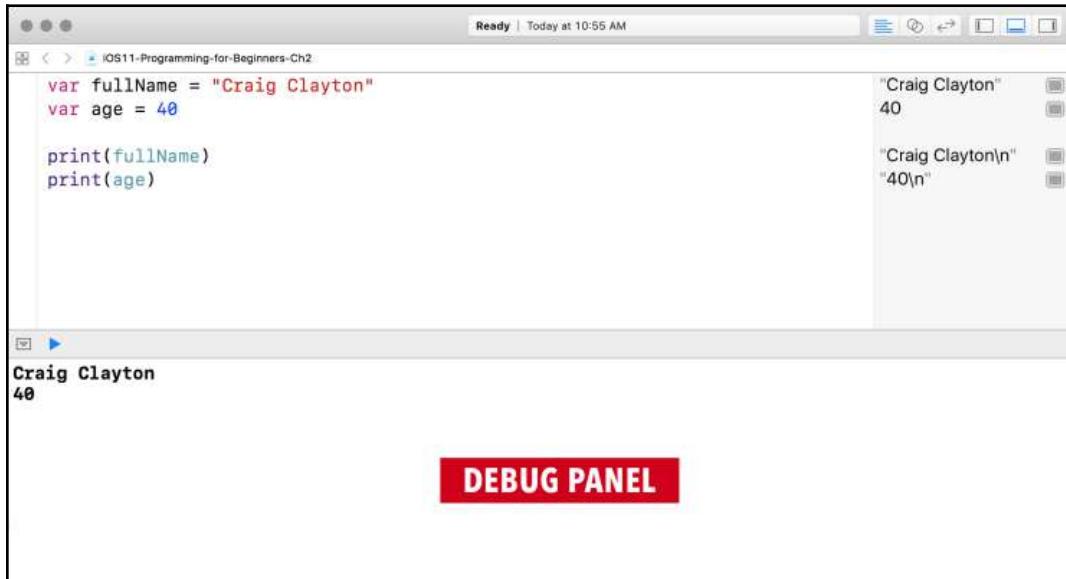


Debug and print() – detecting your bugs

We can use the Debug panel (at the bottom of the following screenshot) using `print()`. So, let's see how `print()` works by printing both our name and age. We can do this by adding the following:

```
print(fullName)  
print(age)
```

It should appear on your screen as follows:



You should now see the output in both the **Results** and **Debug Panels**. Using `print()` allows us to see things in our Debug Panel and therefore verify expected results. Using `print()` is a handy debugging tool.

Adding floating-point numbers

Now let's add floating-point numbers, using the `let` constant, in Playground:

```
let gradeAvg = 2.9  
let version:Float = 1.1
```

This is demonstrated in the following screenshot:

```
var fullName = "Craig Clayton"
var age = 40

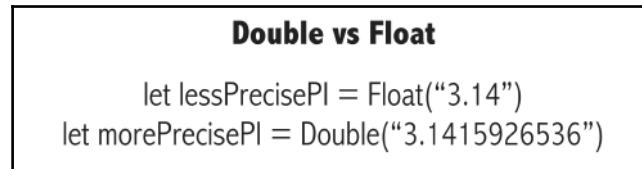
print(fullName)
print(age)

let gradeAvg = 2.9
let version:Float = 1.1
```

"Craig Clayton"	40
"Craig Clayton\n"	"40\n"
2.9	1.1

You will notice that a couple of things are different. First, we are using the `let` keyword. Using `let` tells our program that this is a constant. Constants are variables that cannot change once they are set (as opposed to a non-constant variable, which can change after being set).

The other thing you might have noticed is that we explicitly set our `version` to `Float`. When dealing with a floating-point number, it can be a `Double` or a `Float`. Without getting too technical, a `Double` is much more precise than a `Float`. The best way to explain this is to use pi as an example. Pi is a number in which the digits go on forever. Well, we cannot use a number that goes on forever; however, a `Double` and `Float` handle how precise that number is. Let's look at the following diagram to see what I mean by precise:



So, in the preceding example, you can see that `Float` only displays `3.14`, whereas `Double` gives you a much more accurate number. In Swift, a `Double` is preferred. Therefore, if you do not explicitly set the floating-point number to a `Float`, Swift defaults to a `Double`. To set `version` to a `Float`, you must purposely set it that way.

Creating a Boolean

Now, it is time to create a `Bool`. Let's make it a constant. Enter the following code:

```
let isConstant:Bool = true
```

This is demonstrated in the following screenshot:

The screenshot shows an Xcode playground window titled "Running iOS11-Programming-for-Beginners-Ch2". The code entered is:

```
var fullName = "Craig Clayton"
var age = 40

print(fullName)
print(age)

let gradeAvg = 2.9
let version = 1.1

let isConstant: Bool = true
```

The right side of the window shows the output of the print statements and the value of the constant:

Value
"Craig Clayton"
40
"Craig Clayton\n"
"40\n"
2.9
1.1
true

Since `isConstant` is set, let's make it `false` by adding this:

```
isConstant = false
```

On the same line as what you just entered, you will now see a red circle with a white dot in the middle. The red circle means that there is an error. The white circle inside of it indicates that Xcode can fix the error for you (most of the time):

The screenshot shows an Xcode playground window with the same code as before, but with the last line changed:

```
var fullName = "Craig Clayton"
var age = 40

print(fullName)
print(age)

let gradeAvg = 2.9
let version = 1.1

let isConstant: Bool = true
isConstant = false
```

A red error icon is visible next to the assignment operator (`=`) in the last line. A tooltip message reads: "Cannot assign to value: 'isConstant' is a 'let' constant".

At the bottom of the playground, the error message is displayed:

```
Playground execution failed:
error: iOS11-Programming-for-Beginners-Ch2.playground:5:12: error: cannot assign to value: 'isConstant' is a 'let' constant
isConstant = false
^~~~~~
```

Below that, a note is shown:

```
iOS11-Programming-for-Beginners-Ch2.playground:4:1: note: change 'let' to 'var' to make it mutable
let isConstant: Bool = true
^~~~
```

You will also notice an error in your Debug Panel, which is just a more detailed version of the error. This error is telling us that we are trying to change the value of a constant when we cannot do so.

If you tap on the circle, you will see that Playgrounds suggests that you change the `let` to a `var` since you cannot assign a value to a constant:

The screenshot shows an Xcode playground window titled "iOS11-Programming-for-Beginners-Ch2". The code editor contains the following Swift code:`var fullName = "Craig Clayton"
var age = 40

print(fullName)
print(age)

let gradeAvg = 2.9
let version = 1.1

let isConstant: Bool = true
isConstant = false`The line `isConstant = false` is highlighted with a red arrow pointing to a red callout box. The callout box contains the error message: "Cannot assign to value: 'isConstant' is a 'let' constant" and a suggestion: "Change 'let' to 'var' to make it mutable". A "Fix" button is also visible in the callout box. Below the code editor, the "Playground execution failed" section shows the error output:`error: iOS11-Programming-for-Beginners-Ch2.playground:5:12: error: cannot assign to value: 'isConstant' is a 'let' constant
isConstant = false
^~~~~~`Another red arrow points from the error message in the output to another red callout box containing the same suggestion: "change 'let' to 'var' to make it mutable".

Since we want it to remain a constant, let's delete the line `isConstant = false`. We have covered basic data types, but there are some other programming basics we should discuss as well.

Why constants versus variables?

You might be asking yourself "Why would you ever want to make something constant?". Since constants cannot change after you run your app, they keep you from accidentally breaking a value that should not change. Another excellent use for constants is for base URLs, as you would not want these to change. When you are getting data, you do not want to change the value midway through your app accidentally. Apple recommends that you use `let` whenever possible. Typically, I use `let` until Xcode warns me that a `var` is preferable. If I change the value from `let` to `var`, then I am verifying that this is the behavior I want.

Comments – leaving yourself notes or reminders

Comments are a great way to create notes or reminders to yourself. When you comment code, it means that it will not execute when your code runs. There are two types of comment used: `//` or `/* *\/`. `//` is used for a one-line comment and `/**/` is used for a block of text.

Let's see what both of these look like:

The screenshot shows a Xcode project window titled "iOS11-Programming-for-Beginners-Ch2". The code editor contains the following Swift code:

```
print(fullName)
print(age)

let gradeAvg = 2.9
let version = 1.1

let isConstant: Bool = true

// Single line comment

/*
 This comment is meant for
 multiple lines
 */

Craig Clayton
40
```

To the right of the code, the output is displayed in a table:

Output	Type
"Craig Clayton\n"	String
"40\n"	String
2.9	Double
1.1	Double
true	Bool

The bottom of the Xcode interface shows the command bar with "Craig Clayton" and "40" selected.

Type safety and type inference

Swift is a type-safe language, which means that you are encouraged to be clear about the value types with which your code works. Type inference means that, before your code runs, it quickly checks to ensure that you did not set anything to a different type. If you do, Xcode gives you an error. Why is this good? Let's say that you have an app in the store and that you set one of your variables as a `String` in one part of your code, but then accidentally set the same variable as an `Int` in another part of your code. This error may cause some bad behavior in your app that could cause it to crash. Finding these kinds of errors is like finding a needle in a haystack. Therefore, type checking helps you write safer code by helping you to avoid errors when working with different types.

We have now looked at data types and know that strings are for textual data, `Int` is for integer, `Bool` is for Boolean, and `Double` and `Float` are for floating-point numbers. Let's look a bit deeper into data types and see how we can do more than assign them to variables.

Concatenating strings

String concatenation is the result of combining multiple string literals to form an expression. So, let's create one by first entering two string literals:



A screenshot of the Xcode interface. The title bar says "Ready | Today at 4:49 PM". The main editor area shows the following code:

```
let firstName = "Craig"
let lastName = "Clayton"
```

To the right of the code, there are two preview boxes. The first box contains the string "Craig" and the second box contains the string "Clayton".

Combining these two gives us a string concatenation. We can combine strings by using the `+` operator, add the following:

```
let full = firstName + lastName
```

When you look in the Results Panel, you will notice that there is no space between our first and last names.

Also, if we just put the variables in quotes, they will revert to simple string literals and will no longer be variables.

String interpolation

To correct this, we can put these variables inside quotes, which is known as string interpolation, using a backslash and parentheses around each of our variables inside the string interpolation. Let's update our name variable to the following, and you will see the space in the name in the Results Panel:

```
let full = "\(firstName) \(\lastName)"
```

After adding the preceding line, our code should look something like this:



The screenshot shows the Xcode interface with a single file named "iOS11-Programming-for-Beginners-Ch2". The code editor contains the following Swift code:

```
let firstName = "Craig"
let lastName = "Clayton"

let full = "\(firstName) \(\lastName)"
```

Next to the code editor is the "Results" panel, which displays the output of the print statements:

- "Craig"
- "Clayton"
- "Craig Clayton"

Now that we know about using variables inside quotes, we can do the same inside `print()`. Let's put the `firstName` and `lastName` variables inside `print()`, as follows:

```
print("\(firstName) \(\lastName)")
```

The print statements are great for checking to see whether you are getting the value you want:



A screenshot of an Xcode playground window titled "iOS11-Programming-for-Beginners-Ch2". The code in the editor is:

```
let firstName = "Craig"
let lastName = "Clayton"

let full = "\(firstName) \(lastName)"
print("\(firstName) \(lastName)")
```

The output pane shows the results of the print statements:

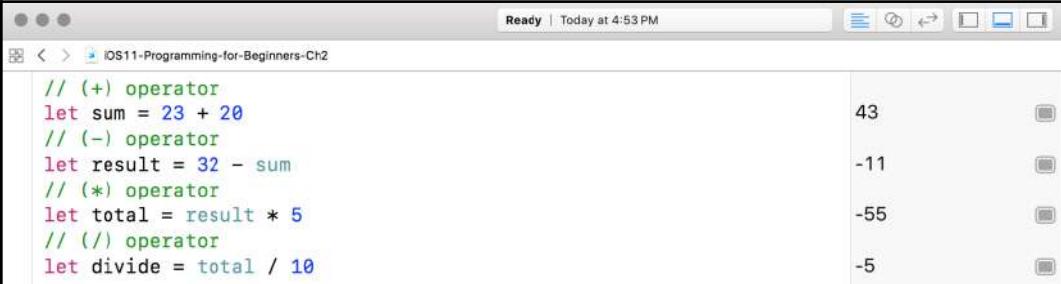
"Craig"	copy
"Clayton"	copy
"Craig Clayton"	copy
"Craig Clayton\n"	copy

The playground's status bar at the bottom shows "Craig Clayton".

Bam! Now, we have a way to view multiple variables inside of `print()` and to create string interpolation by combining multiple strings. We can do much more with Strings, and we will cover that later in this book.

Operations with our integers

In our Playground, we know that age is an int, but with `Int`, we also can write arithmetic expressions using numbers, variables/constants, operators, and parentheses. Let's start with addition, subtraction, multiplication, and division. Add the following into Xcode:



A screenshot of an Xcode playground window titled "iOS11-Programming-for-Beginners-Ch2". The code in the editor is:

```
// (+) operator
let sum = 23 + 20
// (-) operator
let result = 32 - sum
// (*) operator
let total = result * 5
// (/) operator
let divide = total / 10
```

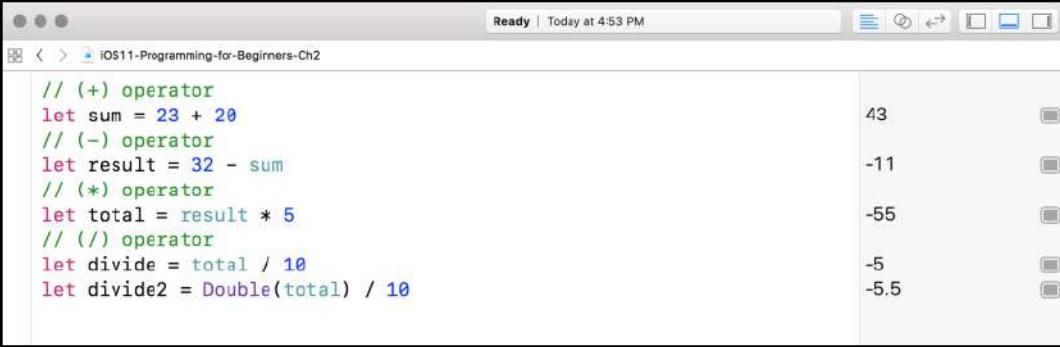
The output pane shows the results of the arithmetic operations:

43	copy
-11	copy
-55	copy
-5	copy

So, sum added two integers (+ operator) together, totaling 43 in our preceding example. Then, we subtracted (- operator) sum from 32 to create a result (-11, in our example). After that, we took the result and multiplied (* operator) it by 5 (see -55 in the Results Panel). All of this is pretty basic math; however, you may have noticed something different with our division equation (/ operator). When you divide two integers, the result is the third integer. So, instead of -55 divided by 10 equals -5.5, our result was -5. To get the correct floating-point value of -5.5, we need to make our division value a Double. Therefore, let's add the following:

```
let divide2 = Double(total) / 10
```

After adding the preceding line of code, your code should look something like this:



The screenshot shows a Xcode Playgrounds interface with a list of operations and their results. The operations are listed on the left, and the results are listed on the right. The results are aligned with the operations they correspond to.

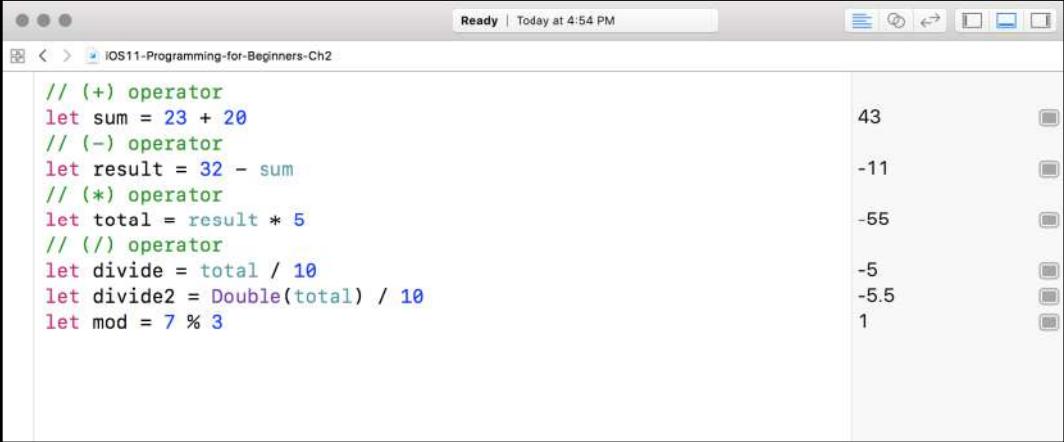
Operation	Result
// (+) operator let sum = 23 + 20	43
// (-) operator let result = 32 - sum	-11
// (*) operator let total = result * 5	-55
// (/) operator let divide = total / 10	-5
let divide2 = Double(total) / 10	-5.5

All of these operations might look familiar to you, but there is one with which you might not be familiar, and that is the remainder operator. The remainder operator returns the remainder when a number is divided by another.

So, for example, 7 divided by 3 equals 2.33. When we apply the remainder operator, we get back 1. Add the following to Playgrounds:

```
let mod = 7 % 3
```

Now, your code should look something like this:



The screenshot shows a Xcode playground window titled "iOS11-Programming-for-Beginners-Ch2". The code entered is:

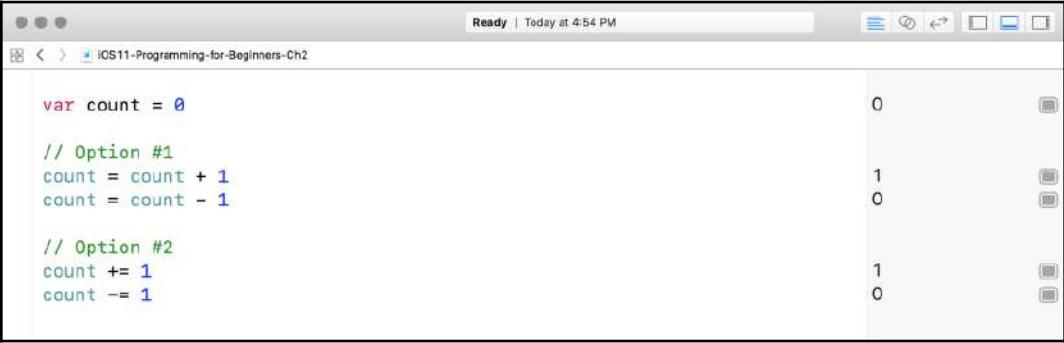
```
// (+) operator
let sum = 23 + 20
// (-) operator
let result = 32 - sum
// (*) operator
let total = result * 5
// (/) operator
let divide = total / 10
let divide2 = Double(total) / 10
let mod = 7 % 3
```

The results column shows the evaluated values for each line:

Line	Value
1	43
2	-11
3	-55
4	-5
5	-5.5
6	1

Increment and decrement

There are times when you need to increment (increase) or decrement (decrease) a value. There are two ways you can accomplish this. Add the following into Playgrounds:



The screenshot shows a Xcode playground window titled "iOS11-Programming-for-Beginners-Ch2". The code entered is:

```
var count = 0

// Option #1
count = count + 1
count = count - 1

// Option #2
count += 1
count -= 1
```

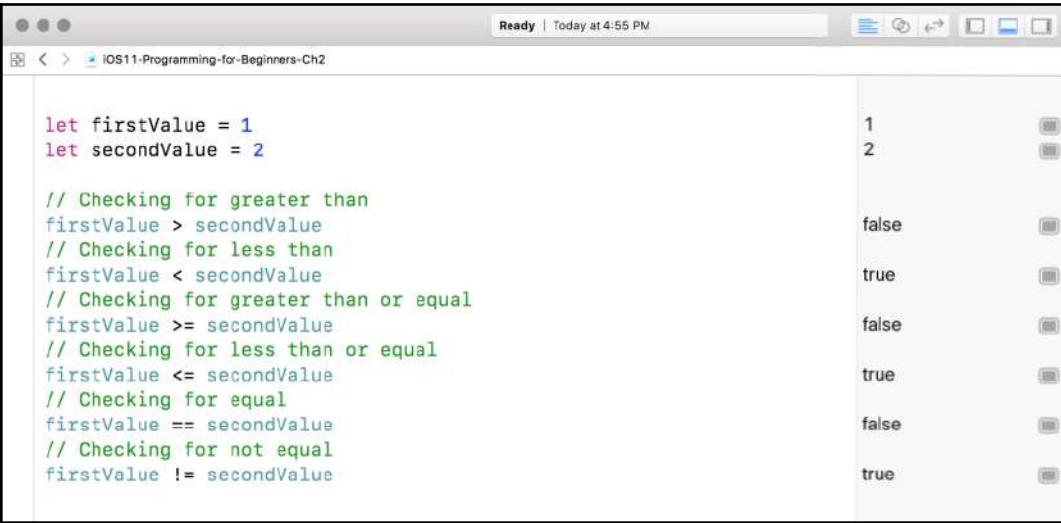
The results column shows the evaluated values for each line:

Line	Value
1	0
2	1
3	0
4	1
5	0

Both of these options do the same thing, but option #2 is in shorthand. The preferred way is to use option #2, which is `+=` (addition assignment operator) and `-=` (subtraction assignment operator), but the choice is yours.

Comparison operators

We can also compare different numerical variables. These might be familiar to you from math class. Let's enter these in to Playgrounds:



The screenshot shows a Xcode Playgrounds interface. In the code editor, two variables are defined: `firstValue = 1` and `secondValue = 2`. Below this, several comparison operations are demonstrated using these values. The results are displayed in the Results panel to the right.

Comparison Operation	Result
<code>firstValue > secondValue</code>	false
<code>firstValue < secondValue</code>	true
<code>firstValue >= secondValue</code>	false
<code>firstValue <= secondValue</code>	true
<code>firstValue == secondValue</code>	false
<code>firstValue != secondValue</code>	true

As you can see in the Results panel, these comparison entries result in true or false based on the values that you enter (here, these are 1 and 2).

Summary

We have hit the basics and, from this point, if you are new to programming, it is a good idea to make sure that you understand each topic we cover. As the chapters progress, we will cover more and more, so take your time and make sure that you are comfortable with all of the topics covered in this chapter.

3

Building on the Swift Foundation

In the last chapter, we went through the basics of understanding data types and how to create variables and constants. Now that we are comfortable with those topics, let's look at adding more building blocks. This chapter will build on what we learned in the previous chapter and get us a bit closer to understanding Swift better.

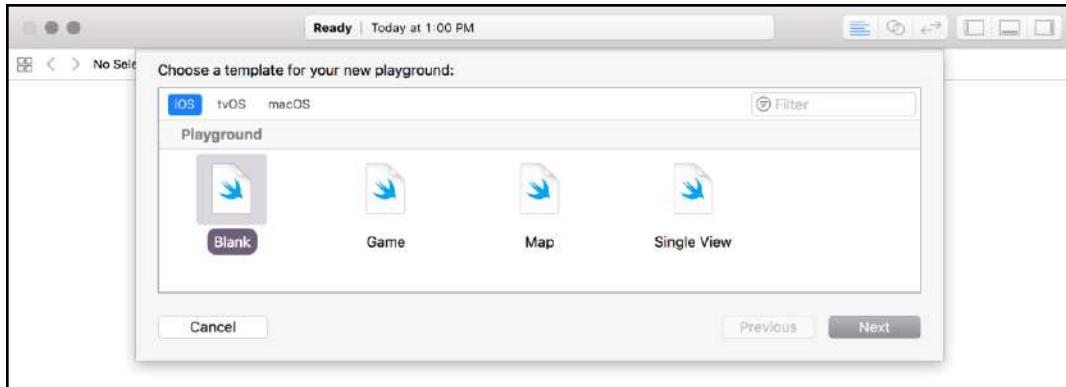
The following topics will be covered in this chapter:

- Type safety and type inference
- Operations with integers
- `if` statements
- Optionals and optional bindings
- Functions

Data types are good, but we will need to add some logic to our app. For example, we want to be able to control whether someone should see a login screen when they launch the app, or whether they should go right into the app. You will use logic a lot, so let's look at what an `if` statement is and how to use it.

Creating a Playground project

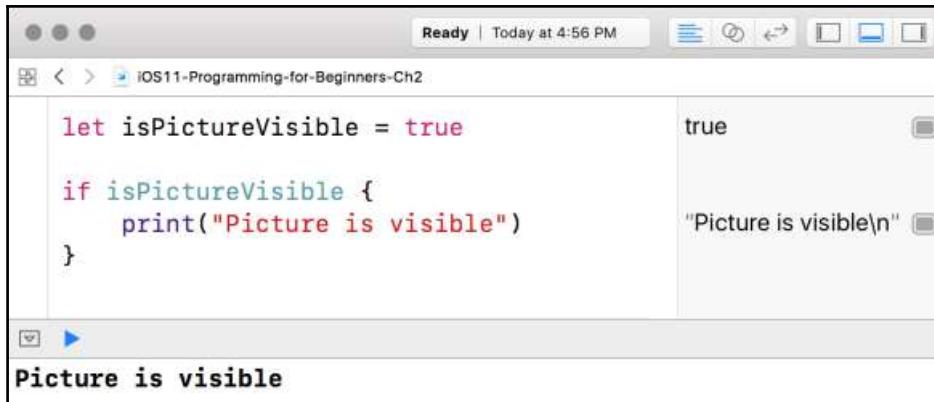
As you learned earlier, launch Xcode and click on **Get started with a playground**:



The Playground template screen will appear. Make sure that you select **iOS** and then choose **Blank** and hit **Next**. You will be asked to give your project a name and a location to save the file; name your new Playground `iOS11-Programming-for-Beginners-Ch3`. You can save the file anywhere you like. Now, with the project saved, we can explore Playgrounds a bit.

The if statements – having fun with logic statements

Let's add our first piece of logic using an `if` statement. An `if` statement is a simple statement to determine whether or not a statement is `true`. Input the following into Xcode:



```
let isPictureVisible = true
if isPictureVisible {
    print("Picture is visible")
}

Picture is visible
```

In the first line of the preceding code, we created a constant named `isPictureVisible`, and we set it to `true`. The next line starts our `if` statement and reads as follows: if `isPictureVisible` is `true`, then print `Picture is visible`. When we write `if` statements, we must use the curly braces to enclose our logic. It is a good practice to put the opening curly brace (`{`) on the same line as the `if` statement and the closing curly brace (`}`) on the line immediately after your logic.

When writing `if` statements using a `bool`, you are always checking for `true`; however, if you wanted to check for `false`, you would do the following:



```
let isRestaurantFound = false
if isRestaurantFound == false {
    print("Restaurant was not found")
}

Restaurant was not found
```

Bools work great with `if` statements, but we also can use them with other data types. Let's try an `if` statement with an `Int` next. Write the following in Playgrounds:

A screenshot of the Xcode Playgrounds interface. On the left, there is a code editor window containing the following Swift code:

```
let drinkingAgeLimit = 19

if drinkingAgeLimit < 21 {
    print("Since we cannot offer you an adult beverage - would you
        like a water or soda to drink?")
}
```

The code editor has a status bar at the bottom with the text "Since we cannot offer you an adult beverage - would you like a water or soda to drink?". To the right of the code editor is a preview pane showing the output of the code. The output shows the value "19" and the printed message "Since we cannot offer you a...". A portion of the message is cut off by an ellipsis.

In the preceding example, we first created another constant with our `Int` set to 19. The next line says: if the `drinkingAgeLimit` is less than 21, then print `Since we cannot offer you an adult beverage - would you like water or soda to drink?`. When you are using `Int` within `if` statements, you will use the comparison operators (`<`, `>`, `<=`, `>=`, `==`, or `!=`). However, our last `if` statement feels incomplete because we are not doing anything for someone over 21. When you need to cover the contradictory statement, this is where you will utilize an `if...else` statement. You enter an `if...else` statement precisely as you do with an `if` statement, but, at the end, you add the word `else`.

You can add `else` to both of the `if` statements we have inputted so far, but, for now, add it to the end of our last `if` statement:

A screenshot of the Xcode Playgrounds interface. On the left, there is a code editor window containing the following Swift code:

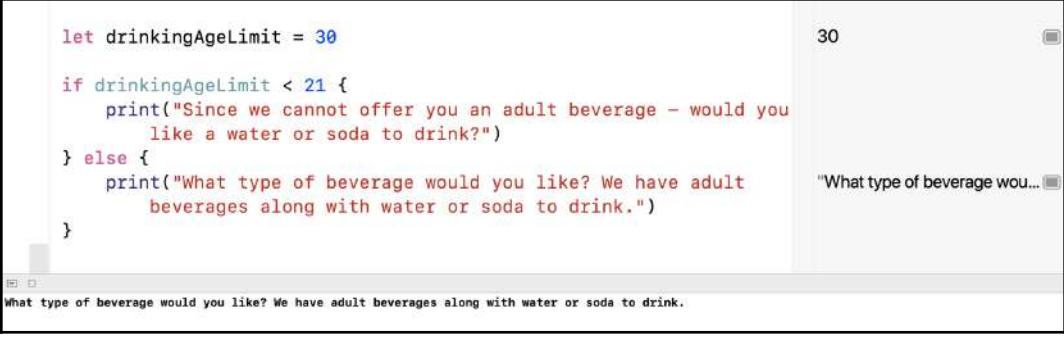
```
let drinkingAgeLimit = 19

if drinkingAgeLimit < 21 {
    print("Since we cannot offer you an adult beverage - would you
        like a water or soda to drink?")
} else {
    print("What type of beverage would you like? We have adult
        beverages along with water or soda to drink.")
}
```

The code editor has a status bar at the bottom with the text "Since we cannot offer you an adult beverage - would you like a water or soda to drink?". To the right of the code editor is a preview pane showing the output of the code. The output shows the value "19" and the printed message "Since we cannot offer you a...". A portion of the message is cut off by an ellipsis.

With else added to the end of our if statement, it turns into an if...else statement, which now reads: if the `drinkingAgeLimit` is less than 21, then print Since we cannot offer you an adult beverage – would you like water or soda to drink? Otherwise (or else), print What type of beverage would you like? We have adult beverages along with water or soda to drink.

Now, our if...else statement can handle both conditions. Based on the value 19 for our `drinkingAgeLimit`, we can see the following in the Debug Panel: Since we cannot offer you an adult beverage – would you like water or soda to drink? If we change `drinkingAgeLimit` to 30, our Debug Panel says What type of beverage would you like? We have adult beverages along with water or soda to drink. Go ahead and change 19 to 30 in Playgrounds:



The screenshot shows the Xcode Playgrounds interface. On the left, there is a code editor window containing the following Swift code:

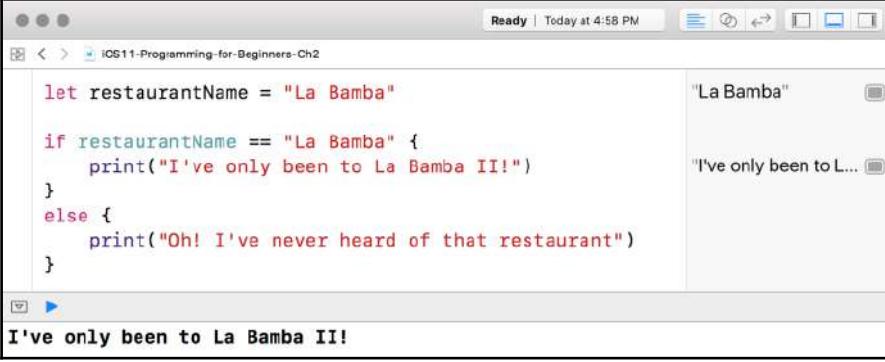
```
let drinkingAgeLimit = 30

if drinkingAgeLimit < 21 {
    print("Since we cannot offer you an adult beverage - would you
        like a water or soda to drink?")
} else {
    print("What type of beverage would you like? We have adult
        beverages along with water or soda to drink.")
}
```

On the right, there is a Debug Panel window. At the top, the variable `drinkingAgeLimit` is set to `30`. Below that, the output of the code is shown in two parts: "What type of beverage wou..." and "What type of beverage wou...". At the bottom of the Debug Panel, the full output is visible: "What type of beverage would you like? We have adult beverages along with water or soda to drink."

Note that we got the behavior we wanted in the Debug Panel.

So far, we have covered using an if statement with a `Bool` and an `Int`. Let's take a look at one more example using a `String`. Add the following bit of code to Playgrounds:



The screenshot shows the Xcode Playgrounds interface. On the left, there is a code editor window containing the following Swift code:

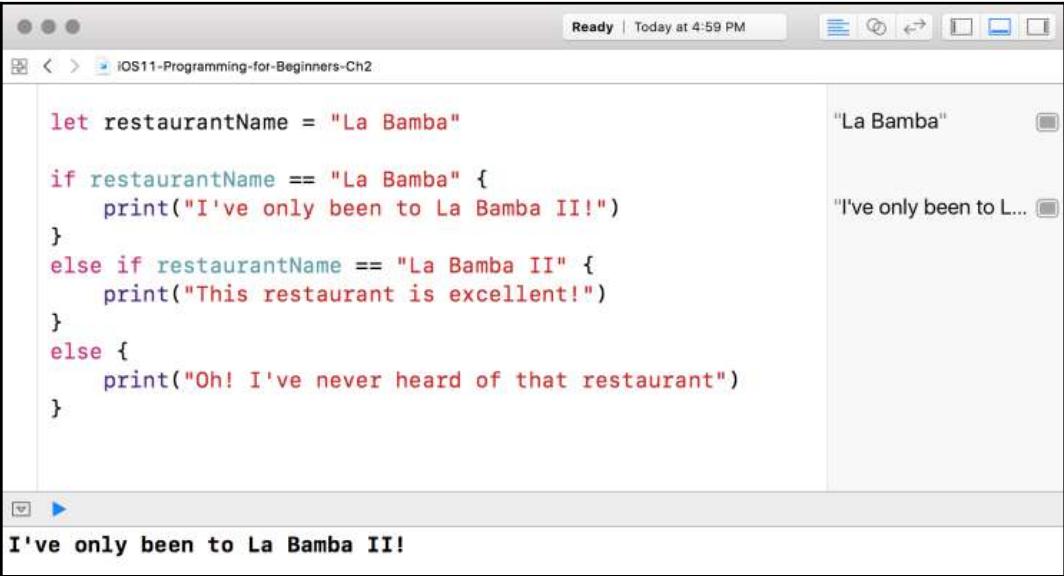
```
let restaurantName = "La Bamba"

if restaurantName == "La Bamba" {
    print("I've only been to La Bamba II!")
} else {
    print("Oh! I've never heard of that restaurant")
}
```

On the right, there is a Debug Panel window. The variable `restaurantName` is set to `"La Bamba"`. The output of the code is shown in two parts: "La Bamba" and "I've only been to L...". At the bottom of the Debug Panel, the full output is visible: "I've only been to La Bamba II!"

In programming, we use equals (=) when setting data to variables. However, to compare two data types, we must use the double equals (==) sign. Therefore, when we write an if statement that compares two strings, we must use double equals (==) instead of a single equal (=) to determine equality.

An if...else statement only lets us check two conditions, whether they are true or not. If we wanted to add more conditions, we would not be able to use just an if...else statement. To accomplish this, we would use what is called an if...else...if...else statement. This statement gives us the ability to add any number of else-if inside our if...else statement. We will not go overboard, so let's add one. Update your last if...else statement to the following:



```
let restaurantName = "La Bamba"

if restaurantName == "La Bamba" {
    print("I've only been to La Bamba II!")
}
else if restaurantName == "La Bamba II" {
    print("This restaurant is excellent!")
}
else {
    print("Oh! I've never heard of that restaurant")
}

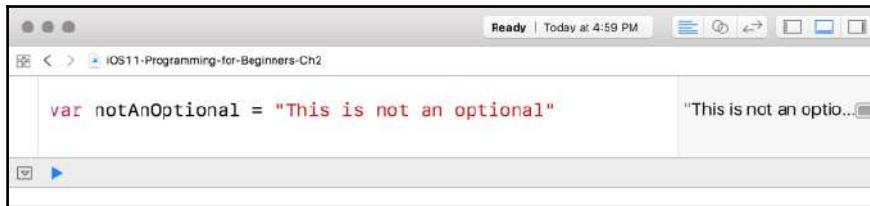
I've only been to La Bamba II!
```

In this example of an if...else...if...else statement, we are checking whether `restaurantName` equals `La Bamba`, then print `I've only been to La Bamba II!`, else if `restaurantName` equals `La Bamba II`, then print `This restaurant is excellent!`, else print `Oh! I've never heard of that restaurant`.

Using if, if...else, and if...else if...else statements helps you to create simple or complex logic for your app. Being able to use them with `Strings`, `bools`, `Ints`, and floating-point numbers give you more flexibility.

Optionals and optional bindings

Use optionals when a value cannot be set. Think of optionals as containers that can take either a value or nil. Optionals gives us the ability to check whether the value is nil or not. To create an optional value, you will have to provide it with a data type followed by a question mark (?). Before we do that, let's create a string that is not an optional. Add the following to Playgrounds:

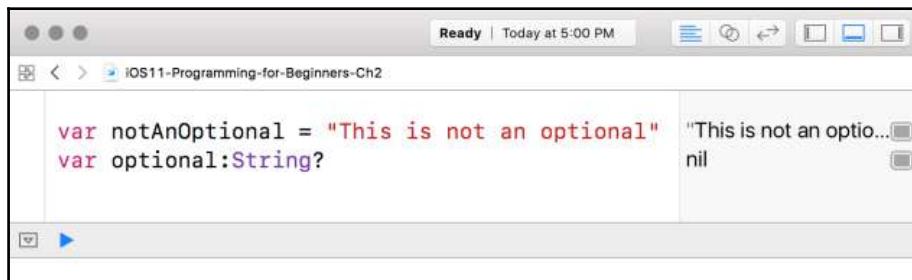


A screenshot of the Xcode Playgrounds interface. The code editor shows the following line of code:

```
var notAnOptional = "This is not an optional"
```

The results panel to the right displays the output: "This is not an optio...".

Now, let's add an optional to Playgrounds:

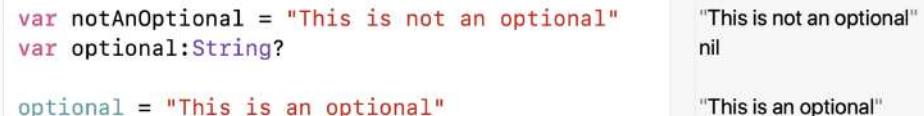


A screenshot of the Xcode Playgrounds interface. The code editor shows two lines of code:

```
var notAnOptional = "This is not an optional"
var optional:String?
```

The results panel to the right shows the output for each line. The first line shows "This is not an optio...". The second line shows "nil".

In this example, we created a string optional, and, if you notice in the Results Panel, it is nil. But for our notAnOptional, we see "This is not an optional". Now, on the next line, let's set optional equal to "This is an optional":



A screenshot of the Xcode Playgrounds interface. The code editor shows three lines of code:

```
var notAnOptional = "This is not an optional"
var optional:String?

optional = "This is an optional"
```

The results panel to the right shows the output for each line. The first line shows "This is not an optional". The second line shows "nil". The third line shows "This is an optional".

In our Results Panel, we see "This is an optional". Now let's print both `notAnOptional` and `optional`, as you will see a difference between the two:

The screenshot shows the Xcode interface with a code editor and a results panel. The code editor contains the following Swift code:var notAnOptional = "This is not an optional"
var optional:String?

print(notAnOptional)
print(optional)

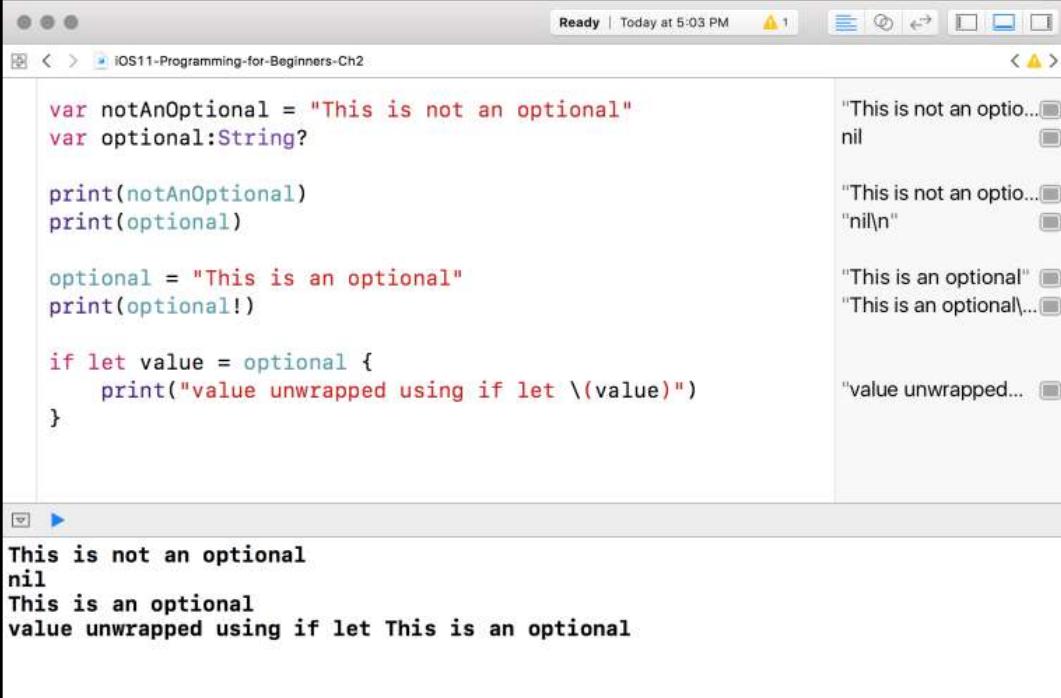
optional = "This is an optional"
print(optional)The results panel shows the output of the print statements. The first two lines are standard strings: "This is not an optional" and "nil". The last two lines show the difference: "This is an optional" and "Optional("This is a...")". Red annotations with arrows point to these lines:

- An arrow points from the text "String is not wrapped inside of an Optional" to the line "This is not an optional".
- An arrow points from the text "Instead of an empty String the value is nil" to the line "nil".
- An arrow points from the text "Notice the String is wrapped inside of the Optional" to the line "Optional("This is an optional")".

Note that our `notAnOptional` variable looks fine, but `optional` has an optional ("") wrapped around the String. The ("") means that, in order for us to access the value, we must unwrap the optional. One way we could do this is by force-unwrapping the optional using an (!). Let's update our print statement and change it to the following:

The screenshot shows the Xcode interface with the same code as before, but the last print statement is updated to use force-unwrapping: `print(optional!)`. The results panel now shows the output: "This is not an optional", "nil", and "This is an optional". The final line "This is an optional" is the result of force-unwrapping the optional value.

We force-unwrapped our optional, but this method is not recommended. If you force-unwrap an optional and there is no value, your app will crash, so avoid this. We should use what is called **optional binding**, which is the safe way to access the value using an `if...let` statement. Remove the `!` from the `print` statement and instead write the following optional binding:



The screenshot shows a Xcode editor window with a Swift file named "iOS11-Programming-for-Beginners-Ch". The code uses optional binding to handle nil values safely.

```
var notAnOptional = "This is not an optional"
var optional: String?

print(notAnOptional)
print(optional)

optional = "This is an optional"
print(optional!)

if let value = optional {
    print("value unwrapped using if let \(value)")
}
```

The output pane shows the results of the code execution:

```
This is not an optional
nil
This is an optional
value unwrapped using if let This is an optional
```

This `if...let` statement is saying that if the optional is not `nil`, set it to `value`, but, if this optional is `nil`, ignore it and do nothing. Now, we do not have to worry about anything regarding setting our value and causing our app to crash.

Why optionals?

So, now, you are probably asking yourself: "Why do you have to do this?". Trust me, when I first learned about optionals, I felt the same way. Using optionals helps to protect your code. For now, understand that when you see a data type followed by a question mark, this variable is an optional. As we work with optionals more and more throughout this book, it will become more evident to you.

Functions

Now, it is time to get into an enjoyable part of programming and learn how to write functions. Functions are self-contained pieces of code that you want to run on something. In Swift 3, Apple made a change to how you should write functions. All of the functions we will write in this chapter will perform an action (think of verbs). Let's create a simple function called `greet()`:

```
func greet() {  
    print("Hello")  
}
```

This example is a basic function with a `print` statement in it. In programming, functions do not run until you call them. We call a function by merely calling its name. So, let's call `greet`, as follows:

```
greet()
```

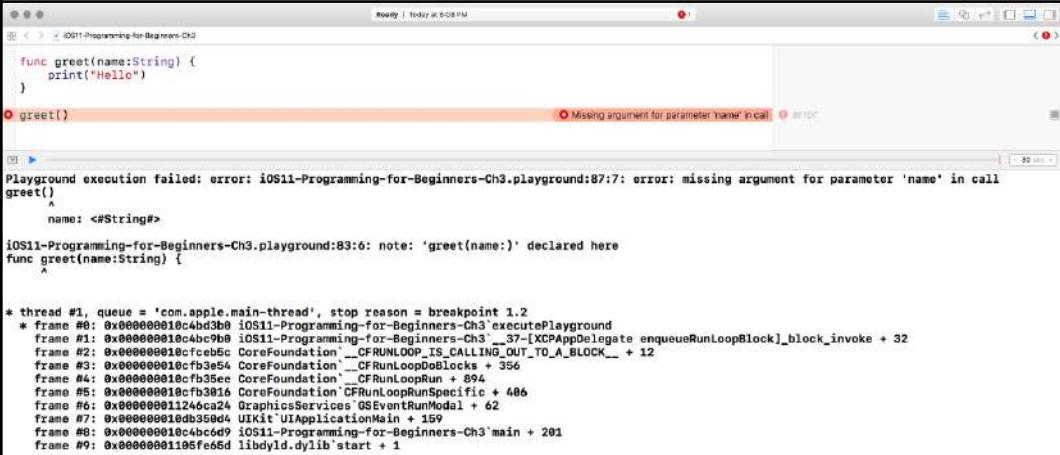
Once we add this to the code, this is what we'll see on screen:



That's it! We just created our first function and called it. However, functions can do so much more. We can add what is called a parameter to a function. A parameter allows us to accept data types inside our parentheses. Doing this will enable us to build more reusable chunks of code. So, let's update our `greet()` function to accept a parameter called `name`:

```
func greet(name:String) {  
    print("Hello")  
}
```

After you update the function, you will get the following error:



The screenshot shows an Xcode playground window titled "iOS11-Programming-for-Beginners-Ch3". In the code editor, there is a single line of code: `greet()`. A red error highlight covers the entire line, with the message "Missing argument for parameter 'name' in call" appearing in the status bar. Below the code editor, the "Playground execution failed" section displays the error message again, along with a note: "note: 'greet(name:)' declared here". The bottom half of the window shows the standard Xcode stack trace for thread #1.

```
func greet(name:String) {
    print("Hello")
}

greet()

Playground execution failed: error: iOS11-Programming-for-Beginners-Ch3.playground:87:7: error: missing argument for parameter 'name' in call
greet()
^
name: <#String#>

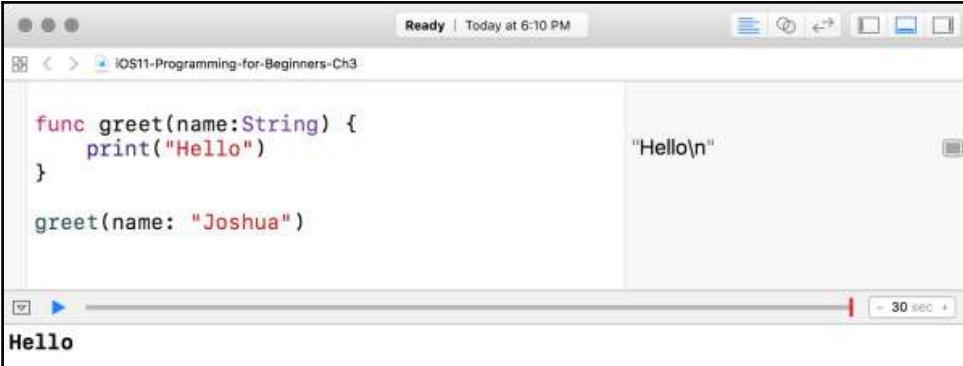
iOS11-Programming-for-Beginners-Ch3.playground:83:6: note: 'greet(name:)' declared here
func greet(name:String) {
^

* thread #1, queue: 'com.apple.main-thread', stop reason = breakpoint 1.2
* frame #0: 0x000000010c4bd3bd iOS11-Programming-for-Beginners-Ch3`executePlayground
frame #1: 0x000000010c4bc9b iOS11-Programming-for-Beginners-Ch3`__37-[XCXAppDelegate enqueueRunLoopBlock]_block_invoke + 32
frame #2: 0x000000010c4bc9b CoreFoundation`_CFAUTOLOCK_IS_CLOSING_OUT_TO_A_BLOCK__ + 12
frame #3: 0x000000010c4bf3e5f CoreFoundation`_CFRunLoopDoBlocks + 356
frame #4: 0x000000010c4fb35ec CoreFoundation`_CFRunLoopRun + 894
frame #5: 0x000000010c4fb3916 CoreFoundation`_CFRunLoopRunSpecific + 486
frame #6: 0x000000010c4fb3a24 GraphicsServices`_GSEventRunModal + 62
frame #7: 0x000000010d4ca24 UIKit`UIApplicationMain + 159
frame #8: 0x000000010c4bcd9 iOS11-Programming-for-Beginners-Ch3`main + 201
frame #9: 0x00000001105fe65d libdyld.dylib`start + 1
```

We received this error because we updated our function, but we did not update the line where we called it. Let's update where we call `greet()` to the following:

```
greet(name: "Joshua")
```

When you are done you should see the following:



The screenshot shows the same Xcode playground window. The code editor now contains two lines: `func greet(name:String) { print("Hello") }` and `greet(name: "Joshua")`. The output pane on the right shows the text "Hello\n". At the bottom left, the word "Hello" is displayed in the console area.

The preceding code looks good; however, the Debug Panel shows us that we are not using the name in our greeting. Earlier, you learned how to create a string interpolation. So, we need to append our variable name inside our `print` statement, as follows:

```
print("Hello \(name)")
```

This is how your code will now look:

The screenshot shows a Xcode interface with a single file named "iOS11-Programming-for-Beginners-Ch3". The code defines a function "greet" that takes a single string parameter "name". Inside the function, it prints "Hello \name". Below the function definition, there is a call to "greet(name: "Joshua")". The output window shows the result: "Hello Joshua\n". At the bottom of the interface, the text "Hello Joshua" is displayed.

```
func greet(name:String) {
    print("Hello \(name)")
}

greet(name: "Joshua")
```

Functions can take multiple parameters, so let's create another `greet()` function that takes two parameters, a first name, and a last name:

```
func greet(first:String, last:String) {
    print("Hello \(first) \(last)")
}
```

Now, your code and its output should look as shown in the following screenshot:

The screenshot shows a Xcode interface with two functions named "greet". The first function takes a single string parameter "name" and prints "Hello \name". The second function takes two string parameters, "first" and "last", and prints "Hello \first \last". Both functions have identical bodies. Below the functions, there is a call to "greet(name: "Joshua")". The output window shows the result: "Hello Joshua\n". At the bottom of the interface, the text "Hello Joshua" is displayed.

```
func greet(name:String) {
    print("Hello \(name)")
}

greet(name: "Joshua")

func greet(first: String, last: String) {
    print("Hello \(first) \(last)")
}
```

We also need to update where we called `greet()` in order to accept multiple parameters as well:

```
greet(first: "Craig", last: "Clayton")
```

Now, your code and output screen should look something like this:

The screenshot shows a Swift code editor with two print statements. The first prints "Hello Joshua\n" and the second prints "Hello Joshua Clayton\n". Below the editor, the terminal window shows the actual output: "Hello Joshua" and "Hello Joshua Clayton".

```
func greet(name:String) {
    print("Hello \(name)")
}

greet(name: "Joshua")

func greet(first: String, last: String) {
    print("Hello \(first) \(last)")
}

greet(first: "Joshua", last: "Clayton")
```

Hello Joshua
Hello Joshua Clayton

We now have a function that takes multiple parameters.

Functions can perform an action, but they can also run an action, and then, when it is done, return back some type of data. Whenever we want our function to return something, we need to use a noun as a way to describe what our function will do. We just created a function called `greet()` that takes a first and last name and creates a full name.

Now, let's create another function called `greeting()`, which will return a full name with a greeting. Let's see what this looks like:

```
func greeting(first:String, last:String) -> String {
    return "Hello \(first) \(last)"
}
```

The following is how your code and output screen should appear:

A screenshot of an Xcode workspace titled "iOS11-Programming-for-Beginners-Ch3". The code editor contains the following Swift code:

```
func greet(name:String) {
    print("Hello \(name)")
}

greet(name: "Joshua")
```

Below the code editor, the console window shows the output:

```
Hello Joshua
```

The output pane also displays the string "Hello Joshua\n".

This function is almost the same as the previous one, but with a couple of new things. First, `-> String` tells the function that we want to return a string. Inside our function, we return `"Hello \(first) \(last)"`. Since we said that we want to return something after our parentheses, we have to do just that. Now, let's look at how we can do this. Enter the following code:

```
print(greeting(first:"Teena", last:"Harris"))
```

Now, this is how your code and output screen should look:

A screenshot of an Xcode workspace titled "iOS11-Programming-for-Beginners-Ch3". The code editor contains the following Swift code:

```
func greet(name:String) {
    print("Hello \(name)")
}

greet(name: "Joshua")
```

Below the code editor, the console window shows the output:

```
Hello Joshua
```

The output pane also displays the string "Hello Joshua\n".

Further down the code editor, there is another function definition:

```
func greeting(with first:String, last:String) -> String {
    return "Hello \(first) \((last))"
}
```

Finally, the code editor shows a call to this function:

```
print(greeting(with: "Teena", last: "Harris"))
```

Below the code editor, the console window shows the output:

```
Hello Teena Harris
```

The output pane also displays the string "Hello Teena Harris\n".

At the bottom of the console window, the text "Hello Teena Harris\n" is also visible.

As you may have noticed, in the Debug Panel, we now have our full name with `Hello` added to the beginning. As you start to build on functions, you start to see the power.

These are just the basics of functions. We will cover more advanced functions throughout our *Let's Eat* app. The main thing novice programmers forget is that functions should be small. Your function should do one thing and one thing only. If your function is too long, then you need to break it up into smaller chunks. Sometimes, longer functions are unavoidable, but you should always be mindful of keeping them as small as possible. Nice work!

Let's work

We covered a lot in this chapter, and now it is time to put everything we covered into practice. Here are two challenges. If you are comfortable with them, then work on them on your own. Otherwise, go back into this chapter, where you can follow along with me and see how you can do each one:

- **Challenge 1:** Write a function that accepts and returns a custom greeting (other than `Hello`, which we addressed earlier in this chapter), along with your first and last name
- **Challenge 2:** Write a function that will take two numbers and add, subtract, multiply, or divide those two numbers

Summary

In this chapter, we learned about operations with integers, as well as working our way through `if` statements. Finally, we discussed the power of optionals and learned about what functions are and how to use them.

In the next chapter, we will move on to some more Swift basics by digging into Swift collections.

4

Digging Deeper

When I first started programming, I was in my mid-twenties. I started a lot older than most, but I will say that grasping the basics took me a bit longer than most, too. I remember when I bought my first programming book, and I read and re-read chapters over and over again until the concepts made sense to me. I found that a lot of books talked to me like I had majored in computer science. As you progress through this book, take your time and, if you need to go back, it is okay to do so. No one is going to care that it took you an extra day to understand a concept. It is more important that you fully understand that concept.

One tip I would give you is to not copy and paste the code. No matter where you find the code and no matter how long it takes, it benefits you to type it out. Doing this benefited me, as I eventually started to remember the code, and it became second nature to me.

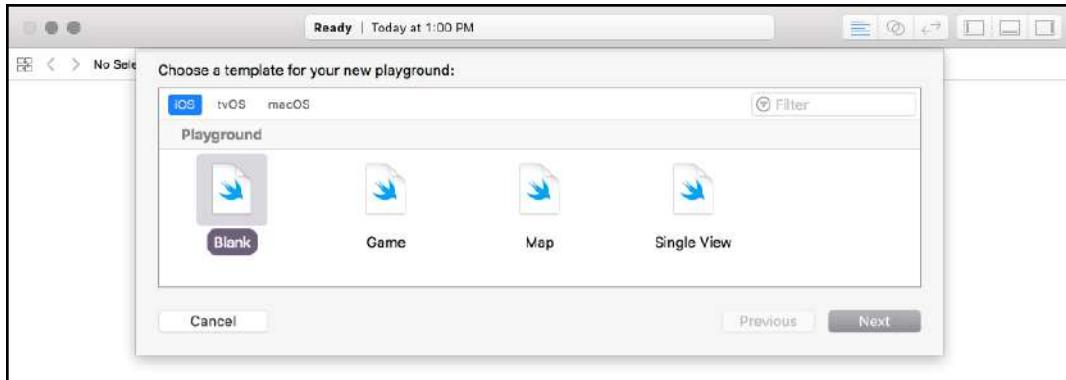
In the last chapter, we went over the basics of Swift to get you warmed up. Now, we will dig deeper and learn about some more programming concepts. These concepts will build on what you have already learned. In this chapter, we will cover the following topics:

- Ranges
- Control flow

Let's begin by creating a new Playground project.

Creating a Playground project

As you learned earlier, launch Xcode and click on **Get started with a playground:**



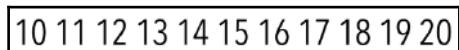
The Playground template screen will appear. Make sure that you select **iOS** and then choose **Blank** and hit **Next**. You will be asked to give your project a name and a location to save the file; name your new Playground `iOS11-Programming-for-Beginners-Ch4`. You can save the file anywhere you like. Now, with the project saved, we can explore Playgrounds a bit.

Next, delete everything inside your file and toggle on the Debug panel using the toggle button (`command + shift + Y`). You should now have a blank screen with the Results Panel on the right, and the Debug Panel opened at the bottom.

We focused on the basics earlier, and now we will build upon those skills. Ranges are one such data type that we should learn about. These are very useful and can come in handy for a variety of reasons. Let's take a look at what ranges are and then start to understand the difference between a *closed range*, a *half-closed range*, and a *one-sided range*.

Ranges

Ranges are generic data types that represent a sequence of numbers. Let's look at the following diagram to understand this:



Closed range

Notice that, in the preceding diagram, we have numbers ranging from **10** to **20**. Rather than having to write each value, we can use ranges to represent all of these numbers in shorthand form. To do this, let's remove all of the numbers in the diagram except for **10** and **20**:

```
10 11 12 13 14 15 16 17 18 19 20
```

Now that we have removed those numbers, we need a way to tell Swift that we want to include all of the numbers that we just deleted. All the numbers in-between are where the range operator (...) comes into play. Therefore, in Playgrounds, let's create a constant called `range` and set it equal to `10...20`:

```
let range = 10...20
```

You should see the following:

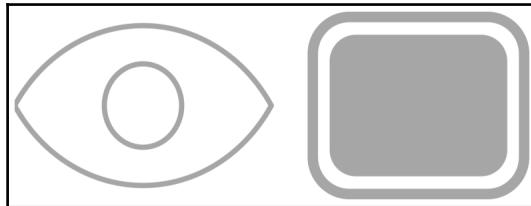


The range that we just entered says that we want the numbers between `10` and `20`, as well as both `10` and `20` themselves. This type of range is known as a closed range.

Inside Playground, in the result, you will see a Show Result icon:



If you hover over the result, you will also see quick look:



Select the Show Result icon so that you can see the result:



We also have what is called a half-closed range.

Half-closed range

Let's make another constant that is known as a half-closed range and set it equal to $10 < 20$. Add the following to Playgrounds:

```
let halfClosedRange = 10..<20
```

Your code should now look like this:

The screenshot shows two separate playgrounds in Xcode's playground editor. The top playground contains the following code:

```
let range = 10...20
lowerBound 10
upperBound 20
```

The bottom playground contains the following code:

```
let halfClosedRange = 10..<20
lowerBound 10
upperBound 20
```

In the results panel, the first playground is labeled "CountableClosedR..." and the second is labeled "CountableRange(1...<20)".

A half-closed range is the same as a closed range, except that the end value is not included. In this example, this means that 10 through 19 are included, and 20 will be excluded.

At this point, you will notice that your Results Panel shows you `CountableClosedRange(10...20)` and `CountableRange(10..<20)`. We cannot see all of the numbers within the range. To see all of the numbers, we need to use a loop.

Control flow

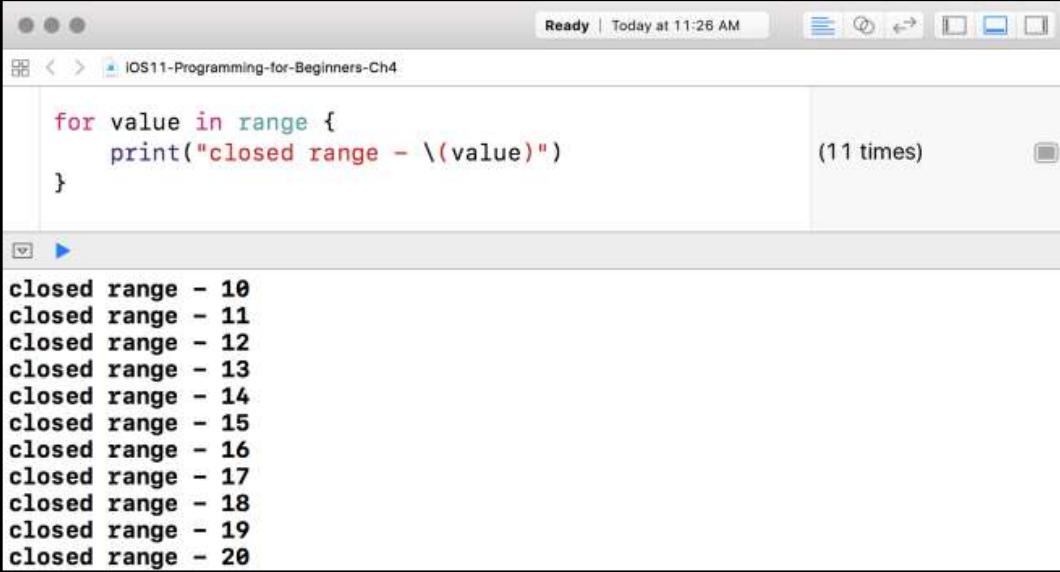
In programming, control flow is the order in which your code executes. When working with Swift, we can use a variety of control statements. Loops, in particular, are useful for when you want to repeat a task multiple times. Let's take a look at a few different types of loop.

The for...in loop

One of the most common control statements is a `for...in` loop. It allows you to iterate over each element in a sequence. Let's see what a `for...in` loop looks like:

```
for <value> in <sequence> {  
    // Code here  
}
```

We start the `for...in` loop with `for`, which is proceeded by `<value>`. `<value>` is a local constant (only the `for...in` loop can access it) and can be any name you like. Typically, you will want to give this value an expressive name. Next, we have `in`, which is followed by `<sequence>`. `<sequence>` is where we want to provide it with our sequence of numbers. Let's write the following into Playgrounds:



A screenshot of the Xcode Playgrounds interface. The title bar says "Ready | Today at 11:26 AM". The code editor shows the following Python-like code:

```
for value in range {  
    print("closed range - \value")  
}
```

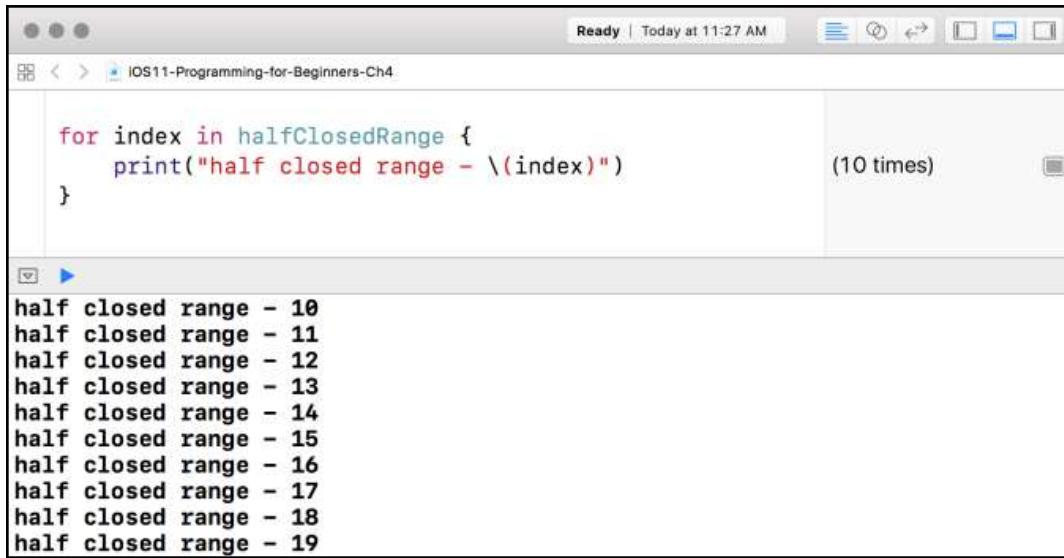
The output pane below shows the results of the loop execution:

```
closed range - 10  
closed range - 11  
closed range - 12  
closed range - 13  
closed range - 14  
closed range - 15  
closed range - 16  
closed range - 17  
closed range - 18  
closed range - 19  
closed range - 20
```

Next to the output, it says "(11 times)".

Notice that, in our Debug Panel, we can see all of the numbers we wanted in our range.

Let's do the same for our `halfClosedRange` variable by adding the following:



The screenshot shows the Xcode Debug Panel. The top bar says "Ready | Today at 11:27 AM". The code editor window has the title "iOS11-Programming-for-Beginners-Ch4". In the code editor, there is a single-line for loop:

```
for index in halfClosedRange {
```

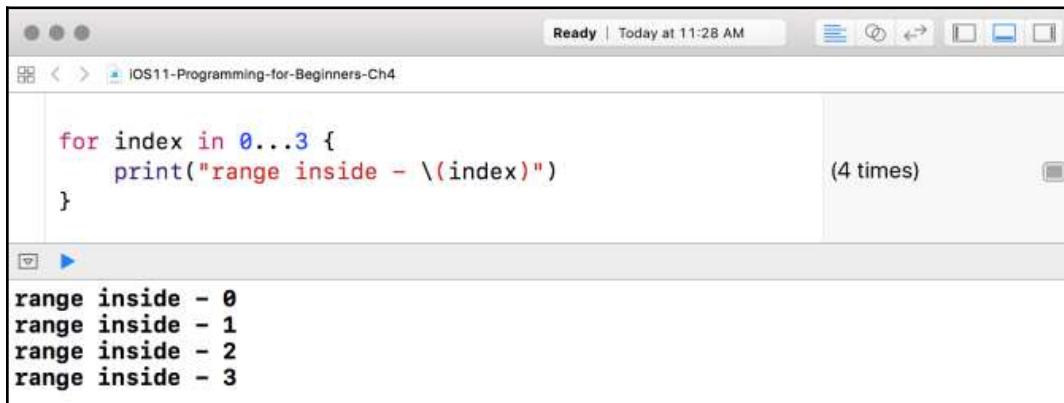
Below the code, the output window shows the results of the print statements:

```
half closed range - 10
half closed range - 11
half closed range - 12
half closed range - 13
half closed range - 14
half closed range - 15
half closed range - 16
half closed range - 17
half closed range - 18
half closed range - 19
```

To the right of the output, it says "(10 times)".

In our Debug Panel, we can see that we get the numbers **10** through **19**. One thing to note is that these two `for...in` loops have different variables. In the first loop, we used the `value` variable, and in the second one, we used the `index` variable. You can make these variables whatever you choose them to be.

Also, in the two preceding examples, we used constants, but we could just use the ranges within the loop. As a next step, you need to add the following:



The screenshot shows the Xcode Debug Panel. The top bar says "Ready | Today at 11:28 AM". The code editor window has the title "iOS11-Programming-for-Beginners-Ch4". In the code editor, there is a single-line for loop:

```
for index in 0...3 {
```

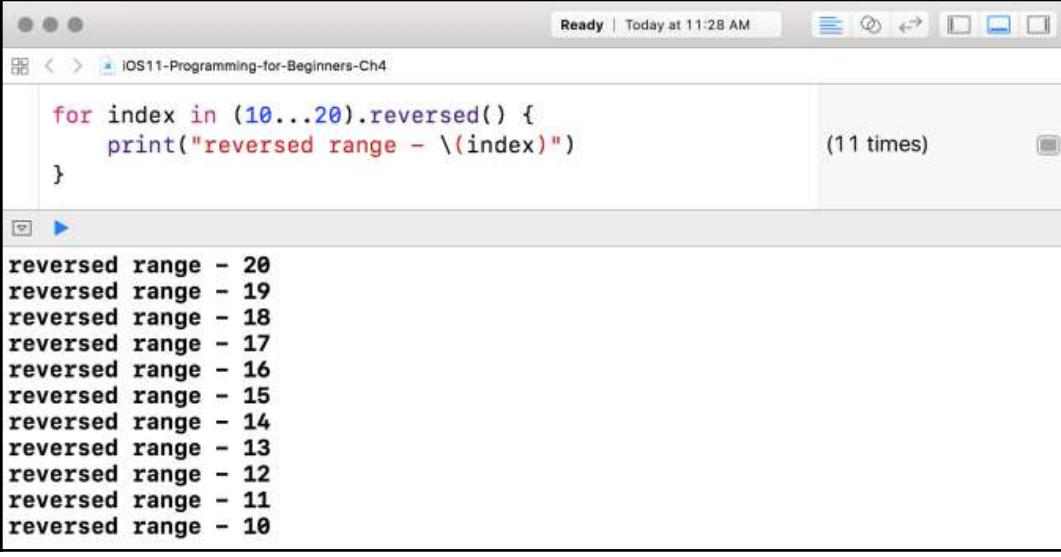
Below the code, the output window shows the results of the print statements:

```
range inside - 0
range inside - 1
range inside - 2
range inside - 3
```

To the right of the output, it says "(4 times)".

Now, you will see **0** to **3** print inside the Debug Panel.

What if you wanted the numbers to go in reverse order? Let's input the following for...in loop:



The screenshot shows a Xcode interface with a code editor and a terminal window. The code editor contains the following Python code:

```
for index in (10...20).reversed():
    print("reversed range - \\" + str(index) + ")")
```

The terminal window shows the output of the code, which is:

```
reversed range - 20
reversed range - 19
reversed range - 18
reversed range - 17
reversed range - 16
reversed range - 15
reversed range - 14
reversed range - 13
reversed range - 12
reversed range - 11
reversed range - 10
```

The terminal status bar indicates "Ready | Today at 11:28 AM".

One-sided range

A one-sided range operator allows you to use ranges that continue as far as possible in one direction. If you want the range to continue, then this is what you would use. Let's look at a one-sided range by adding the following:

```
let names = ["Craig", "Teena", "Jason", "Joshua", "Myah", "Tiffany",
"Kim", "Veronica", "Mikki(KK)", "Milan", "Shelby", "Kaysey"]

for name in names[2...]:
    print(name)
}
```

You will see that all the names print in the console:



The screenshot shows a Xcode project window titled "iOS11-Programming-for-Beginners-Ch4". The main area contains the following Swift code:

```
let names = ["Craig", "Teena", "Jason", "Joshua", "Myah", "Tiffany", "Kim", "Veronica", "Mikki(KK)", "Milan", "Shelby", "Kaysey"]

for name in names[2...] {
    print(name)
}
```

The output of the code is displayed in the "Console" tab, showing the names from index 2 to 11:

```
Jason
Joshua
Myah
Tiffany
Kim
Veronica
Mikki(KK)
Milan
Shelby
Kaysey
```

A status bar at the top right indicates "(10 times)".

As a next step, let's add the following:

```
for name in names[...6] {
    print(name)
}

// Craig
// Teena
// Jason
// Joshua
// Myah
// Tiffany
// Kim
```

You should now see in the console how this update changes what is in the console:



The screenshot shows a Xcode playground window titled "iOS11-Programming-for-Beginners-Ch4". In the code editor, there is a single line of Swift code:

```
for name in names[...6] {  
    print(name)  
}
```

Below the code, the output pane shows the names printed seven times:

```
Craig  
Teena  
Jason  
Joshua  
Myah  
Tiffany  
Kim
```

A status bar at the top right indicates "(7 times)" next to the print icon.

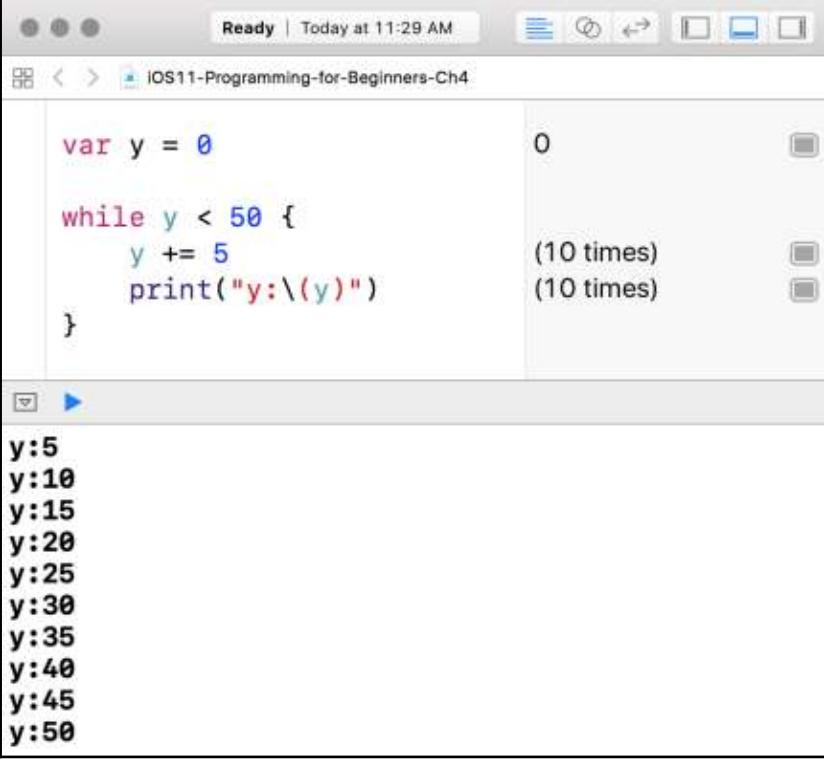
Another useful loop is the `while` loop. Let's take a look at how the `while` loop is used.

The `while` loop

A `while` loop executes a Boolean expression at the start of the loop, and the set of statements run until a condition becomes `false`. It is important to note that `while` loops can execute zero or more times. Here is the basic syntax of a `while` loop:

```
while <condition> {  
    // statement  
}
```

Let's write a while loop in Playgrounds and see how it works. You have to add the following:



The screenshot shows the Xcode Playgrounds interface. At the top, there are three circular status indicators, the word "Ready", the date and time "Today at 11:29 AM", and several icons for file operations. Below this is a toolbar with icons for back, forward, search, and other playground controls. The main area is titled "iOS11-Programming-for-Beginners-Ch4". In the code editor, the following Swift code is shown:

```
var y = 0
while y < 50 {
    y += 5
    print("y: \(y)")
}
```

Below the code, the output window displays the results of the loop iteration:

```
y:5
y:10
y:15
y:20
y:25
y:30
y:35
y:40
y:45
y:50
```

So, this while loop starts with a variable that begins at zero. Before the while loop executes, it checks to see whether `y` is less than 50, and, if so, it continues into the loop. Using the `+=` operator, which we covered earlier, we increment `y` by five each time. Our while loop will continue to do this until `y` is no longer less than 50. Now, let's add the same while loop after the one we created and see what happens:

```
while y < 50 {
    y += 5
    print("y: \(y)")
}
```

When you are done, you should see the following:

```
Ready | Today at 11:29 AM
```

```
var y = 0
```

```
while y < 50 {
```

```
    y += 5
```

```
    print("y:\\"y")
```

```
}
```

```
while y < 50 {
```

```
    y += 5
```

```
    print("y:\\"y")
```

```
}
```

```
y:5
```

```
y:10
```

```
y:15
```

```
y:20
```

```
y:25
```

```
y:30
```

```
y:35
```

```
y:40
```

```
y:45
```

```
y:50
```

You will notice that the second `while` loop never runs. This may not seem like it is essential until we look at our next type of loop.

The repeat...while loop

The `repeat...while` loop is pretty similar to a `while` loop in that it continues to execute the set of statements until a condition becomes `false`. The main difference is that the `repeat...while` loop does not evaluate its bool condition until the end of the loop. Here is the basic syntax of a `repeat...while` loop:

```
repeat {  
    // statement  
} <condition>
```

Let's write a `repeat...while` loop in Playgrounds and see how it works. Add the following to Playgrounds:

```
var x = 0

repeat {
    x += 5
    print("x: \(x)")
} while x < 100

print("repeat completed x: \(x)")
```

The screenshot shows the Xcode Playgrounds interface with the title "Ready | Today at 11:30 AM". The code area contains the provided Swift code. The results pane shows the output of the print statements. The variable `x` starts at 0 and increases by 5 each iteration, reaching 100. The final output is "repeat completed x: 100".

Output	Description
var x = 0	Initial declaration of variable <code>x</code> .
repeat {	Start of the <code>repeat</code> block.
x += 5	Increment <code>x</code> by 5.
print("x: \(x)")	Print the value of <code>x</code> .
} while x < 100	End of the <code>repeat</code> block, followed by the condition <code>while x < 100</code> .
print("repeat completed x: \(x)")	Final print statement after the loop completes.
x: 5 x: 10 x: 15 x: 20 x: 25 x: 30 x: 35 x: 40 x: 45 x: 50 x: 55 x: 60 x: 65 x: 70 x: 75 x: 80 x: 85 x: 90 x: 95 x: 100	Output of the <code>x</code> values from the loop iterations.
repeat completed x: 100	Final output message.

You will notice that our `repeat...while` loop executes first and increments `x` by 5, and after, as opposed to checking the condition like it did before, as with a `while` loop, it checks to see whether `x` is less than 100. This means that our `repeat...while` loop will continue until the condition hits 100. Here is where it gets interesting.

Let's add another `repeat...while` loop after the one we just created:

```
repeat {
    x += 5
    print("x: \(x)")
} while x < 100
```

The screenshot shows an Xcode project window titled "iOS11-Programming-for-Beginners-Ch4". The code editor contains the following Swift code:

```
var x = 0

repeat {
    x += 5
    print("x: \(x)")
} while x < 100

print("repeat completed x: \(x)")

repeat {
    x += 5
    print("x: \(x)")
} while x < 100
```

The output pane displays the results of the execution:

Output	Description
0	Initial value of x
(20 times)	Number of iterations for the first loop
(20 times)	Number of iterations for the second loop
"repeat completed..."	Text printed after the first loop
105	Final value of x after the second loop
"x: 105\n"	Text printed after the second loop

The console pane shows the individual values of x for each iteration of the inner loop:

```
x: 5
x: 10
x: 15
x: 20
x: 25
x: 30
x: 35
x: 40
x: 45
x: 50
x: 55
x: 60
x: 65
x: 70
x: 75
x: 80
x: 85
x: 90
x: 95
x: 100
repeat completed x: 100
x: 105
```

Now, you can see that our `repeat...while` loop incremented to 105 instead of 100, like the previous `repeat...while` loop. This happens because the bool expression does not get evaluated until after it is incremented by 5. Knowing this behavior will help you to pick the correct loop for your situation.

Summary

So far, we have looked at three loops: the `for...in` loop, the `while` loop, and the `repeat-while` loop. We will use the `for...in` loop again, but first, we need to talk about collections.

In the next chapter, we will focus on what collections are and how to use them when working with data. Make sure that you fully understand loops, because we will build on them in the next chapter and throughout this book. Therefore, review as much as you need so that you feel confident that you are proficient in the topics contained in this chapter.

5

Digging into Collections

In the last couple of chapters, we reviewed the basics of Swift to get you warmed up. Before we start building our app, we need to look at one more programming concept—collections. In Swift, we have three primary collection types, which we will cover in this chapter:

- Arrays
- Dictionaries
- Sets

We will dig deeper into each one, but we will start with the most common collection type—arrays.

Arrays

Arrays are ordered collections of values and can hold any number of items, for example, a list of strings, ints, and floating-point values. Arrays are stored in an ordered list, starting at 0. Let's look at a diagram:

0	Florida	0	45	0	Florida
1	Ohio	1	66	1	California
2	California	2	23	2	32
3	North Carolina	3	10	3	New York
4	Colorado	4	88	4	99
5	Nevada			5	true
6	New York			6	9.0

Starting from left to right in the preceding examples, we first have an array that holds a collection of strings. In the second example, we have another array that holds a collection of ints. In our third example, we have an array that holds a collection of mixed data values.

Now, let's review the following diagram, which is a mixed array:

0	Florida
1	California
2	32
3	New York
4	99
5	true
6	9.0

Since this example contains mixed data types, such as strings, ints, and bools, we would have to name this an array type of Any. This means that we can have mixed data types inside our array. Until you are genuinely comfortable with arrays, I would not recommend using mixed data arrays. Try to stick to arrays with the same data type because you will know the exact data type of each element.

An array can hold any data type, but making the array strongly typed means that every element in it must be of the same type.

Creating an empty array

Now let's create a few arrays in Playgrounds.

Sometimes, you may want to remove your prior entries from your Playground so that it is easier for you to see each new `print` statement. Do that now and input the following:



The screenshot shows an Xcode playground window titled "Running iOS11-Programming-for-Beginners-Ch5". The code area contains the following Swift code:

```
let integers:[Int] = []
let strings = [String]()
```

We just created our first two arrays. The reason for two different syntaxes is because you can create it in two different ways. Regarding the first example, you will see me create arrays throughout this book.

The data types within each set of brackets tells Swift what type of array we want to create. The first array (`integers`) we created has a data type of ints, and our second array (`strings`) has a data type of strings.

Creating an array with initial values

Arrays can have initial values when they are created. Let's see how this would look by entering the following in Playgrounds:

```
let integers2 = [54, 29]
```

Your code will now look like this:



The array that we just entered uses type inference to declare the data type of the array using its initial values. In this case, Swift understands that it is an array of ints because the values we entered are integers. In addition, when we use a constant (`let`) on an array, we are telling Swift that the array is an immutable array, which means that the contents or size cannot change once it is instantiated.

Creating a mutable array

It is a best practice to make all arrays (and, for that matter, collections) immutable, but there are some cases where you will need to create an array that is mutable. Let's have some fun and create a mutable array:

```
var states:[String] = []
```

As an aside, when creating a mutable array (or any variable), note that each variable must be unique.

One use of a mutable array is so that we can add to the array. Let's look at some ways in which we can do this.

Adding items to an array

Let's add some data to our array. There are a few different convenience methods for adding data to an array.

A convenience method is, just as its name implies, a method that makes things convenient. A method is a function that lives inside a class. We will discuss classes later in this book. If this is starting to get overwhelming, that's understandable. You do not need to worry about every single detail at this time. We will cover this again, and things will slowly start to click at some point. Everyone learns differently, so there is no reason to worry if someone else understands something more quickly. Just go at your own pace.

The first convenience method we will look at is the `append()` method:

```
states.append(23)
```

Your code and the output window should now look like this:

The screenshot shows an Xcode playground window titled "iOS11-Programming-for-Beginners-Ch5". In the code editor, the following Swift code is written:

```
let integers:[Int] = []
let strings = [String]()
let integers2 = [54, 29]
var states:[String] = []

states.append(23)
```

An error message is displayed in the status bar: "Cannot convert value of type 'Int' to expected argument type 'String'". Below the code editor, the output window shows the error message:

```
Playground execution failed:

error: iOS11-Programming-for-Beginners-Ch5.playground:1:15: error: cannot convert value of type 'Int' to expected argument
type 'String'
states.append(23)
^~
```

Houston, we have a problem! You will see that we are getting an error. I did this for a couple of reasons, because getting errors is normal and common. Most people who start out coding are afraid to make a mistake or get scared about getting or seeing errors. Trust me, I have been coding for years, and I make mistakes all the time. The error is telling us that we tried to add an int into an array that can only hold strings.

For every developer, whether they are a beginner or a veteran, there will come a time when they encounter an error that they cannot figure out. This error might get you frustrated to the point where you want to throw the computer across the room (I have been there a few times). The best advice my boss ever gave me was to take a walk for 10-15 minutes or do something to take your mind off it. Sometimes, this helps, and you will come up with an idea after you walk away. Even if you come back and it still takes you hours to figure out what is wrong, this is still part of the process. The best errors are the ones where you overlooked the simplest thing and had to spend hours trying to figure it out. You might have lost time, but you will have learned a great lesson. Lessons like these will stay with you forever, and you will never forget the error the next time you encounter it. So, if your coding results in an error, even in this book, embrace the challenge, because there is no greater feeling than figuring out a challenging error.

So, let's correct what we just did by revising the array to show the following:

```
states.append("Florida")
```

The following is how your code should now look:

A screenshot of an Xcode workspace titled "iOS11-Programming-for-Beginners-Ch5". The playground contains the following Swift code:

```
let integers:[Int] = []
let strings = [String]()
let integers2 = [54, 29]
var states:[String] = []

states.append("Florida")
0 "Florida"
```

To the right of the code, there is a table view displaying the state of each variable:

Variable	Type	Value
integers	[Int]	[]
strings	[String]	[]
integers2	[Int]	[54, 29]
states	[String]	[]
states	[String]	["Florida"]
0	[String]	"Florida"

In the Results Panel, you will see the contents of our corrected array.

Since an array can hold any number of items, let's add some more. Earlier, I mentioned that we have a variety of ways to add items to an array. The `append()` method allows us to add only one item at a time. To add multiple items, we can use the convenience called `append(contentsOf:)`.

Add the following to Playgrounds:

```
states.append(contentsOf: ["California", "New York"])
```

Now, your code should look like this:

The screenshot shows the Xcode interface with a playground document titled "iOS11-Programming-for-Beginners-Ch5". In the code editor, the following Swift code is written:

```
let integers:[Int] = []
let strings = [String]()
let integers2 = [54, 29]
var states:[String] = []

states.append("Florida")
0 "Florida"

states.append(contentsOf: ["California", "New York"])
0 "Florida"
1 "California"
2 "New York"
```

To the right of the code editor, the "Results" pane displays the state of the `states` array after each operation:

- After `states.append("Florida")`: `[["Florida"]]`
- After `states.append(contentsOf: ["California", "New York"])`: `[["Florida", "California", "New York"]]`

We added two more items to our array, but, so far, every example we have utilized has added items at the end of our array. We have two convenience methods that allow us to add items at any index position that is available in the array.

The first method we can utilize to do this is called `insert(at:)`, which allows us to add a single item at a specific index position. We also have `insert(contentsOf:at:)`, which enables us to add multiple items into an array at a certain index position. Let's use them both and add `Ohio` after `Florida` and then `North Carolina`, `South Carolina`, and `Nevada` in front of `New York`:

```
states.insert("Ohio", at:1)
states.insert(contentsOf:[ "North Carolina", "South Carolina",
    "Nevada"], at:3)
```

Now, your code should look like this:

The screenshot shows the Xcode interface with a project titled "iOS11-Programming-for-Beginners-Ch5". In the main editor, there is some Swift code defining arrays for integers, strings, and states, and then using the `append()` and `insert()` methods on the `states` array. To the right of the code, there is a sidebar showing the current state of the `states` array as an array of strings. The array starts with "Florida", followed by "California" and "New York" (which are collapsed), then "Ohio", and finally "North Carolina", "South Carolina", and "Nevada" (which are also collapsed). The sidebar also shows the original contents of the array before modifications.

```
let integers:[Int] = []
let strings = [String]()
let integers2 = [54, 29]
var states:[String] = []

states.append("Florida")
0 "Florida"

states.append(contentsOf: ["California", "New York"])
0 "Florida"
1 "California"
2 "New York"

states.insert("Ohio", at:1)
0 "Florida"
1 "Ohio"
2 "California"
3 "New York"

states.insert(contentsOf:[ "North Carolina", "South Carolina", "Nevada"], at:3)
0 "Florida"
1 "Ohio"
2 "California"
3 "North Carolina"
4 "South Carolina"
```

We just added items to our array using `append(contentsOf:)`, but there also is a shorthand version of this, which is done by using the `+ =` operator. As a next step, let's add the following:

```
states += [ "Texas", "Colorado" ]
```

Now, your code should look like this:

The screenshot shows a Xcode editor window with several code snippets demonstrating array mutation methods. The code is written in Swift.

```
Ready | Today at 11:57 AM
```

```
iOS11-Programming-for-Beginners-Ch5
```

```
let integers:[Int] = []
let strings = [String]()
let integers2 = [54, 29]
var states:[String] = []

states.append("Florida")
0 "Florida"

states.append(contentsOf: ["California", "New York"])
0 "Florida"
1 "California"
2 "New York"

states.insert("Ohio", at:1)
0 "Florida"
1 "Ohio"
2 "California"
3 "New York"

states.insert(contentsOf:["North Carolina", "South Carolina", "Nevada"], at:3)
0 "Florida"
1 "Ohio"
2 "California"
3 "North Carolina"
4 "South Carolina"

states += ["Texas", "Colorado"]
0 "Florida"
1 "Ohio"
2 "California"
3 "North Carolina"
```

This technique for adding items is much more concise and is my preferred way of inserting items into an array. Writing less code is not always better but, in this case, using the `+ =` operator is my go-to method.

Checking the number of elements in an array

If you are keeping track, we now have nine items in our array. Luckily, we do not have to keep track of how many items are in our array because we have a property called `count`. This property will keep track of the current item count and give us the total count of our array when we want to check. Let's look at the `count` for states:

```
states.count
```

Your code will now look like this:

The screenshot shows a Xcode editor window with the title bar "Ready | Today at 11:57 AM" and the file name "iOS11-Programming-for-Beginners-Ch5". The code in the editor is as follows:

```
let integers:[Int] = []
let strings = [String]()
let integers2 = [54, 29]
var states:[String] = []

states.append("Florida")
0 "Florida"

states.append(contentsOf: ["California", "New York"])
0 "Florida"
1 "California"
2 "New York"

states.insert("Ohio", at:1)
0 "Florida"
1 "Ohio"
2 "California"
3 "New York"

states.insert(contentsOf:["North Carolina", "South Carolina", "Nevada"], at:3)
0 "Florida"
1 "Ohio"
2 "California"
3 "North Carolina"
4 "South Carolina"

states += ["Texas", "Colorado"]
0 "Florida"
1 "Ohio"
2 "California"
3 "North Carolina"
4 "South Carolina"

states.count
9
```

The right side of the editor shows the state of the arrays after each operation:

- After `states.append("Florida")`: `[54, 29]`, `["Florida"]`
- After `states.append(contentsOf: ["California", "New York"])`: `["Florida", "California", "New York"]`
- After `states.insert("Ohio", at:1)`: `["Florida", "Ohio", "California", "New York"]`
- After `states.insert(contentsOf:["North Carolina", "South Carolina", "Nevada"], at:3)`: `["Florida", "Ohio", "North Carolina", "South Carolina", "Nevada"]`
- After `states += ["Texas", "Colorado"]`: `["Florida", "Ohio", "North Carolina", "South Carolina", "Nevada", "Texas", "Colorado"]`
- After `states.count`: `9`

Checking for an empty array

The `count` property is not the only property we can use to calculate how many items are in an array. The most commonly used property for an array is called `isEmpty`. This property uses the `count` property by checking to see whether the count is greater than 0. This method will return `true` or `false`, depending on whether there are any items within our array. Since you learned that `if...else` statements work well with bools, let's use this `isEmpty` property in an `if...else` statement.

Add the following to Playgrounds:

```
if states.isEmpty {  
    print("There are no items in the array")  
}  
else {  
    print("There are currently \(states.count) total items in our  
array")  
}
```

Now, your code and output should look like this:

The screenshot shows the Xcode Playgrounds interface. On the left, the playground code is displayed in Swift:

```
let integers:[Int] = []  
let strings = [String]()  
let integers2 = [54, 29]  
var states:[String] = []  
  
states.append("Florida")  
states.append(contentsOf: ["California", "New York"])  
states.insert("Ohio", at:1)  
states.insert(contentsOf:["North Carolina", "South Carolina", "Nevada"], at:3)  
states += ["Texas", "Colorado"]  
states.count  
  
  
if states.isEmpty {  
    print("There are no items in the array")  
}  
else {  
    print("There are currently \(states.count) total items in our array")  
}
```

On the right, the playground output is shown in a table format:

[]	
[]	
[54, 29]	
[]	
["Florida"]	
["Florida", "Californ...	
["Florida", "Ohio", "...	
["Florida", "Ohio", "...	
["Florida", "Ohio", "...	
9	

Below the playground area, the console output is displayed:

```
There are currently 9 total items in our array
```

Now, our Debug panel prints the following: There are currently total 9 items in our array.

One thing to remember in programming is that occasionally, there are multiple ways of writing a piece of code. It is not shocking to meet someone who will approach the same problem differently to you. To me, this is why programming is so amazing. Ultimately, all that matters is that it works as expected, especially when you are new to programming.

All programming languages have what is known as a style guide, which is a preferred way of writing code, and it is no different in Swift. Using the preferred style means a suggested method, but even then, you will notice that most preferred methods differ on certain things. For now, you do not need to worry about different style guides, other than to know that they exist. In this book, we will follow a style that I have incorporated into my code.

Once you get comfortable, I recommend that you start to look at style guides and adapt them into your code. Knowing different styles helps you to know your options as well as to understand what others are doing with their code, even if you do not agree with how they write something. If you write your code with a defined structure or style throughout a project, it will make it easier for you to come back to your code if you, for instance, had to take a break for some reason, such as starting another project, or just taking some time off.

Retrieving a value from an array

We discussed creating arrays as well as adding items to an array. Now, let's turn to retrieving a value from an array. Since arrays are stored by their index, we can use their index to retrieve values. By way of an example, let's retrieve California:

```
let state = states[3]
```

Now, your code should look like this:

The screenshot shows a Xcode playground window titled "iOS11-Programming-for-Beginners-CH5". The code in the playground is as follows:

```
let integers:[Int] = []
let strings = [String]()
let integers2 = [54, 29]
var states:[String] = []

states.append("Florida")
states.append(contentsOf: ["California", "New York"])
states.insert("Ohio", at:1)
states.insert(contentsOf:["North Carolina", "South Carolina", "Nevada"], at:3)
states += ["Texas", "Colorado"]
states.count

if states.isEmpty {
    print("There are no items in the array")
} else {
    print("There are currently \(states.count) total items in our array")
}

let state = states[3]
print(state)
```

The right side of the window shows the results of the code execution:

- Integers: []
- Strings: []
- Integers2: [54, 29]
- States (array): ["Florida"]
- States (array): ["Florida", "California", "New York"]
- States (array): ["Florida", "Ohio", "North Carolina", "South Carolina", "Nevada"]
- States (array): ["Florida", "Ohio", "North Carolina", "South Carolina", "Nevada", "Texas", "Colorado"]
- States (count): 9
- If block output: "There are currently..." (truncated)
- Let state output: "North Carolina"

The Results Panel shows North Carolina and not California. Remember, arrays start at 0, not 1. Therefore, for us to get California, we would actually need to use the index position of 2. Let's make that update in Playgrounds as follows:

```
let state = states[2]
```

When you are done, you should see that we get "California":

The screenshot shows the Xcode Playgrounds interface. On the left, the code is written in Swift:

```
let integers:[Int] = []
let strings = [String]()
let integers2 = [54, 29]
var states:[String] = []

states.append("Florida")
states.append(contentsOf: ["California", "New York"])
states.insert("Ohio", at:1)
states.insert(contentsOf:["North Carolina", "South Carolina", "Nevada"], at:3)
states += ["Texas", "Colorado"]
states.count

if states.isEmpty {
    print("There are no items in the array")
} else {
    print("There are currently \(states.count) total items in our array")
}

let state = states[2]
California
```

On the right, the Results panel displays the output of the code:

- [54, 29]
- ["Florida"]
- ["Florida", "Californi..."]
- ["Florida", "Ohio", "..."]
- ["Florida", "Ohio", "..."]
- ["Florida", "Ohio", "..."]
- 9
- "There are currently..."
- "California"

There we go!

We now have this great list of states, but someone told you that Arizona is also amazing. Instead of just adding Arizona to our list, you decide that you'd actually prefer to replace South Carolina with Arizona. We could simply look at our array and see in which index South Carolina is located. This would not be helpful, however, if it were to change, or if the state for which you were searching did not exist. So, the safe way to code this is to check the array for an item, and, if that item is found, then Swift will give us its current index position. The `index(of:)` method is what we will use to get the index position of South Carolina:

```
if let index = states.index(of:"South Carolina") {  
    print("Current index position is \(index)")  
}
```

This is how our code and output should now appear:

The screenshot shows a Swift playground in Xcode. The code in the playground is as follows:

```
let integers:[Int] = []  
let strings = [String]()  
let integers2 = [54, 29]  
var states:[String] = []  
  
states.append("Florida")  
states.append(contentsOf: ["California", "New York"])  
states.insert("Ohio", at:1)  
states.insert(contentsOf:["North Carolina", "South Carolina", "Nevada"], at:3)  
states += ["Texas", "Colorado"]  
states.count  
  
  
if states.isEmpty {  
    print("There are no items in the array")  
}  
else {  
    print("There are currently \(states.count) total items in our array")  
}  
  
let state = states[2]  
California  
  
if let index = states.index(of: "South Carolina") {  
    print("Current index position of South Carolina is \(index)")  
}
```

The right side of the interface shows the state of the arrays and the output of the print statements:

- `integers`: []
- `integers2`: [54, 29]
- `states`: ["Florida", "California", "New York", "Ohio", "North Carolina", "South Carolina", "Nevada", "Texas", "Colorado"]
- Output:
 - "There are currently 9 total items in our array"
 - "Current index position of South Carolina is 4"

Now that we have the position, we can replace South Carolina with Arizona, as follows:

```
if let index = states.index(of:"South Carolina") {  
    states[index] = "Arizona"  
}
```

This is how our code should now look:

The screenshot shows the Xcode interface with a project titled "iOS11-Programming-for-Beginners-Ch5". In the editor, the code has been modified to include the replacement of South Carolina with Arizona:

```
let integers:[Int] = []  
let strings = [String]()  
let integers2 = [54, 29]  
var states:[String] = []  
  
states.append("Florida")  
states.append(contentsOf: ["California", "New York"])  
states.insert("Ohio", at:1)  
states.insert(contentsOf:["North Carolina", "South Carolina", "Nevada"], at:3)  
states += ["Texas", "Colorado"]  
states.count  
  
if states.isEmpty {  
    print("There are no items in the array")  
}  
else {  
    print("There are currently \(states.count) total items in our array")  
}  
  
let state = states[2]  
California  
  
if let index = states.index(of: "South Carolina") {  
    states[index] = "Arizona"  
}
```

The right side of the screen shows the Debug Panel with the following output:

- [54, 29]
- [{"Florida"}, {"Florida", "California", "New York"}, {"Florida", "Ohio", "North Carolina", "South Carolina", "Nevada"}, {"Florida", "Ohio", "North Carolina", "South Carolina", "Nevada", "Texas", "Colorado"}]

Count: 9

"There are currently..."

"California"

"Arizona"

Iterating over an array

It would be nice if we could see a list of states in our array. Earlier, you learned that `for...in` loops work with sequences. Since our array is a sequence, we can use `for...in` loops to loop through each element. When working on a project that has arrays, it is helpful to use a `print` statement inside a `for...in` loop. This lets us print every item in our array to the Debug Panel. So, let's use a `for...in` loop to look at the contents of our array:

```
for state in states {  
    print(state)  
}
```

This is how our code and output should now look:



```
for state in states {
    print(state)
}

Florida
Ohio
California
North Carolina
Arizona
Nevada
New York
Texas
Colorado
```

The screenshot shows a terminal window with a light gray background. In the top right corner, there is a small square icon with a minus sign. The code is written in a monospaced font. The output of the code is listed below it, showing each state name on a new line. The text is black and has a slight shadow effect.

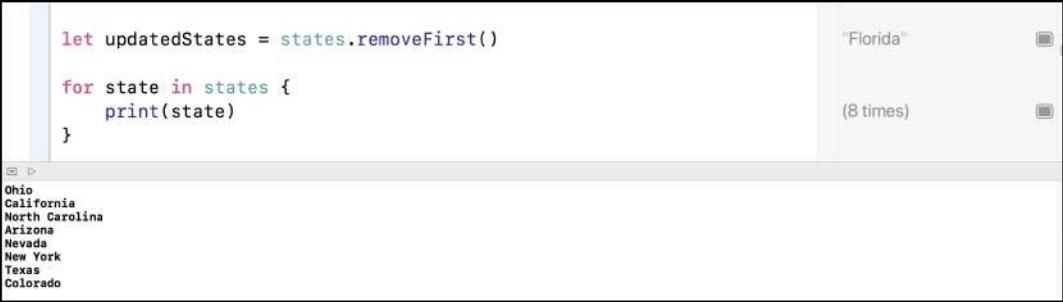
Removing items from an array

Now, it is time to start deleting items from our array. Let's delete the first item from our list. We have a convenience method for removing items from an array, called `removeFirst()`. This method will remove the first item from our array, which, in our case, is Florida. Let's remove Florida and add this line above our `for...in` loop:

```
let updatedStates = states.removeFirst()

for state in states {
    print(state)
}
```

This is how our code and output should now look:



```
let updatedStates = states.removeFirst()

for state in states {
    print(state)
}

"Florida"
(8 times)

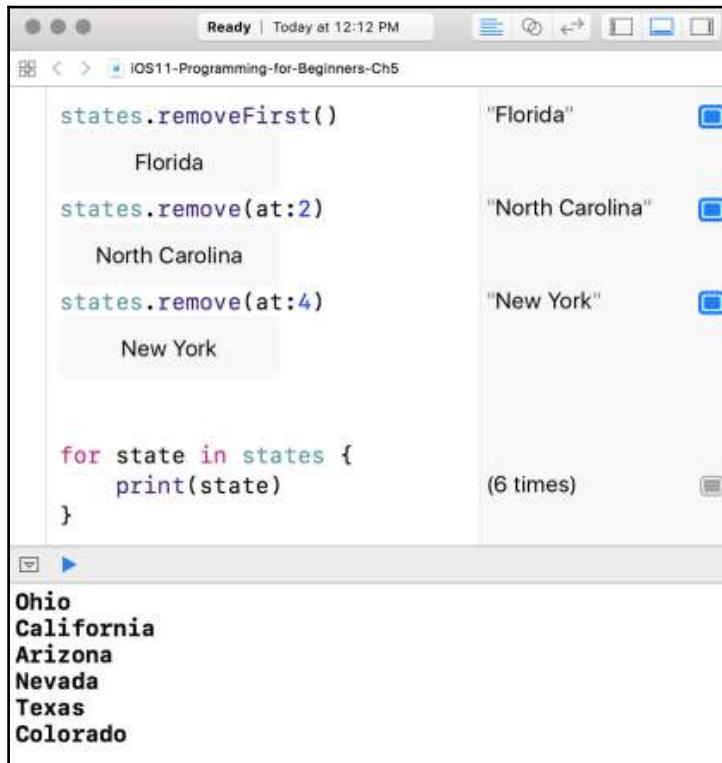
Ohio
California
North Carolina
Arizona
Nevada
New York
Texas
Colorado
```

The screenshot shows a terminal window with a light gray background. In the top right corner, there are two small square icons with minus signs. The code is identical to the previous one. The output shows the word "Florida" followed by "(8 times)" in parentheses, indicating it was printed 8 times. Below this, the list of states is shown again, starting with Ohio and ending with Colorado.

Since we removed Florida, all of our states' index positions will be updated to move one position closer to the top of the array. But what if we wanted to remove an item that was not first? To do this, we can use the `remove(at:)` convenience. So, let's remove North Carolina and New York, which are sitting at positions 2 and 4, respectively. We will add the following above our `for...in` loop:

```
states.remove(at:2)
states.remove(at:4)
```

This is how our code and output should now look:



The screenshot shows the Xcode Playgrounds interface. The code area contains the following:

```
states.removeFirst()
Florida
states.remove(at:2)
North Carolina
states.remove(at:4)
New York

for state in states {
    print(state)
}
```

The output area shows the results of the code execution:

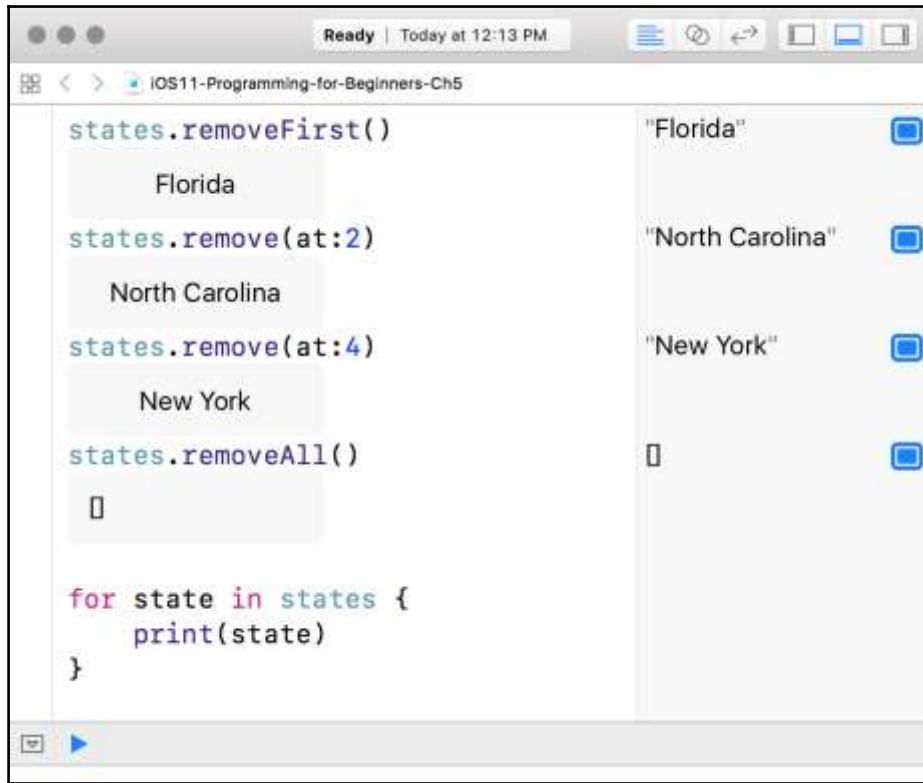
```
Ohio
California
Arizona
Nevada
Texas
Colorado
```

The output shows that Florida, North Carolina, and New York have been removed from the list. The remaining states (Ohio, California, Arizona, Nevada, Texas, Colorado) are printed in sequence.

Now, both North Carolina and New York have been removed. You will see that California and Ohio did not move, but Colorado and Nevada moved up closer to the top of the list. To remove the remaining six items, we could use `remove(at:)` for each one, but instead we will use the simpler method of `removeAll()`. So, let's use `removeAll()` in Playgrounds:

```
states.removeAll()
```

Now, your code should look something like this:



The screenshot shows a Swift playground in Xcode. The code in the playground is as follows:

```
states.removeFirst()
Florida
states.remove(at:2)
North Carolina
states.remove(at:4)
New York
states.removeAll()
[]

for state in states {
    print(state)
}
```

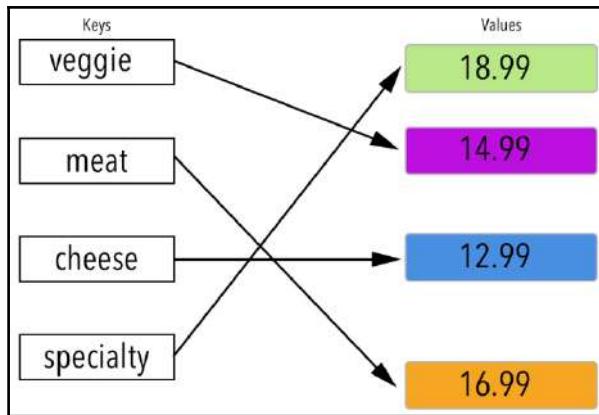
The output pane shows the results of the code execution:

Code	Output	Action
<code>states.removeFirst()</code>		
Florida	"Florida"	checkbox
<code>states.remove(at:2)</code>		
North Carolina	"North Carolina"	checkbox
<code>states.remove(at:4)</code>		
New York	"New York"	checkbox
<code>states.removeAll()</code>	[]	checkbox
<code>for state in states {</code>		
print(state)		
}		

Now, we are back to where we started with an empty array. We have only scratched the surface of arrays. We will do more with arrays later in this book, but we first need to look at the next collection type: dictionaries.

Dictionaries

A dictionary is an unordered collection of values, with each one accessed through a unique key. Let's look at the following diagram:



In our diagram, we have a dictionary of pizzas (**keys**) with their prices (**values**). To find something inside a dictionary, we must look it up according to its key. Let's look at a dictionary syntax:

```
Dictionary<Key, Value>
```

Now, that we understand what a dictionary is and its syntax let's look at how we can use it by creating our first dictionary.

Creating a dictionary

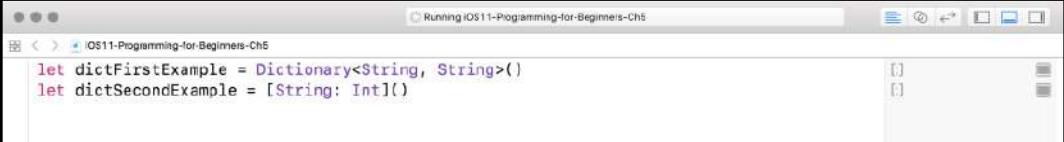
The traditional way of creating a dictionary is to first declare it as a dictionary and then, inside angle brackets, declare a type for the key and value. Let's create our first dictionary inside Playgrounds:

```
Ready | Today at 12:31 PM
File ▾ ▶ iOS11-Programming-for-Beginners-Ch5
let dictFirstExample = Dictionary<String, String>()
```

The immutable dictionary we created earlier has a string data type for both its key and value. We have multiple ways to create a dictionary. Let's look at another by adding the following to Playgrounds:

```
let dictSecondExample = [String: Int]()
```

Your code should now look like this:



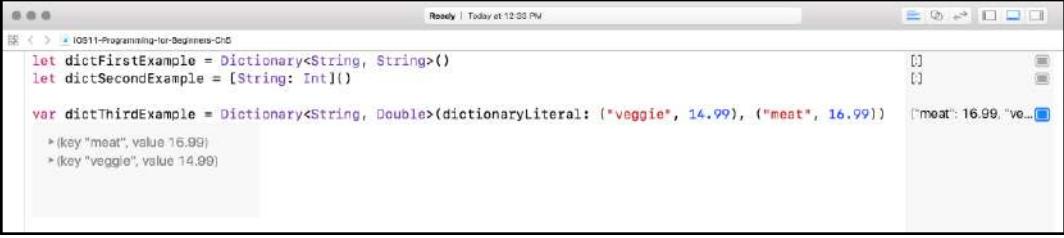
```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()
```

In this latest example, we created another immutable dictionary, with its key having a string data type and its value having an int data type.

If we wanted to use our pizza diagram, the key would have a string data type and the value would have a double data type. Let's create this dictionary in Playgrounds, but, this time, we will make it a mutable dictionary and give it an initial value:

```
var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral:
("veggie", 14.99), ("meat", 16.99))
```

Your code should now look like this:



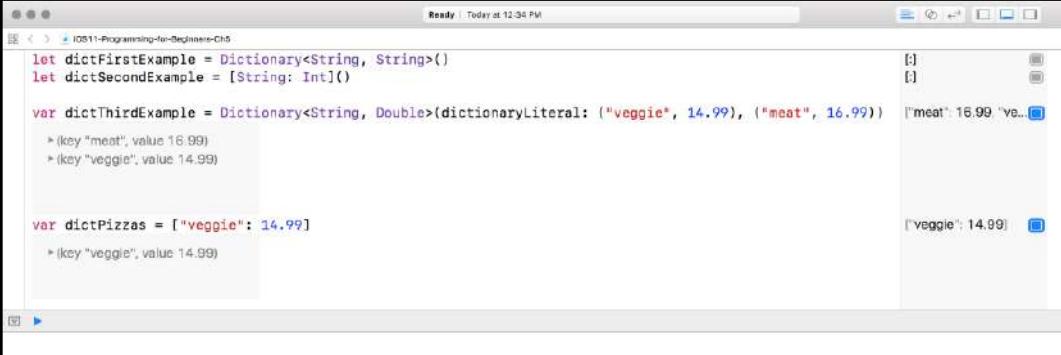
```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ("veggie", 14.99), ("meat", 16.99))
  ↴ (key "meat", value 16.99)
  ↴ (key "veggie", value 14.99)
```

The preceding example is just one way of creating a dictionary for our pizza diagram example. Let's look at a much more common method using type inference:

```
var dictPizzas = ["veggie": 14.99]
```

Once you add this to your code, your code should look something like this:



The screenshot shows a Xcode editor window with a single item in the `dictPizzas` dictionary. The code is as follows:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ("veggie", 14.99), ("meat", 16.99))
  ▶ (key "meat", value 16.99)
  ▶ (key "veggie", value 14.99)

var dictPizzas = ["veggie": 14.99]
  ▶ (key "veggie", value 14.99)
```

The right pane shows the state of the dictionary:

```
[{"veggie": 14.99}]
```

The preceding is a much simpler way of creating a dictionary with an initial value. When initializing a dictionary, it can have any number of items. In our case, we are starting off with just one.

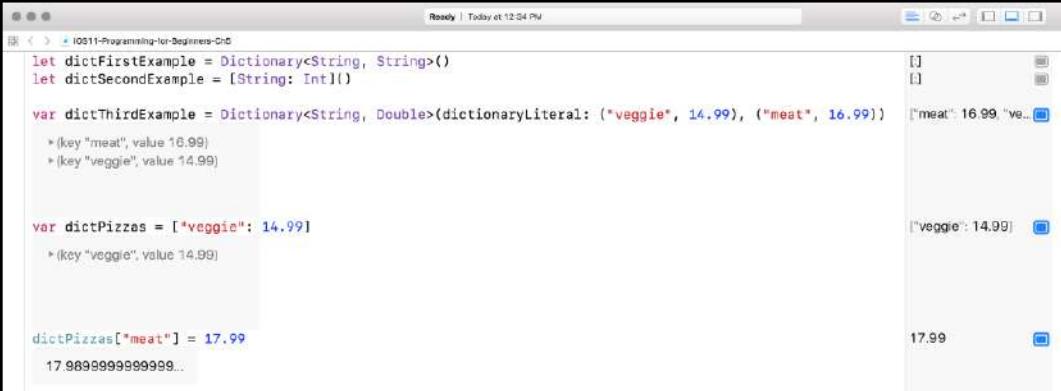
Now, let's look at how we can add more pizzas to our dictionary.

Adding and updating dictionary elements

Let's add another item to our `dictPizzas` dictionary:

```
dictPizzas["meat"] = 17.99
```

Once you add this line of code, your code snippet should look like this:



The screenshot shows a Xcode editor window with two items in the `dictPizzas` dictionary. The code is as follows:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ("veggie", 14.99), ("meat", 16.99))
  ▶ (key "meat", value 16.99)
  ▶ (key "veggie", value 14.99)

var dictPizzas = ["veggie": 14.99]
  ▶ (key "veggie", value 14.99)

dictPizzas["meat"] = 17.99
  17.99
```

The right pane shows the state of the dictionary:

```
[{"veggie": 14.99}, {"meat": 17.99}]
```

This is the shorthand method for adding an item to a dictionary. After the dictionary variable, we add the key inside the brackets. Since the key for this dictionary is strings, we must put this key in quotes. Next, we assign a double to our value. Now, our dictionary has two items. This syntax is also used to update a dictionary item. Let's change the price of meat pizza to 16.99:

```
dictPizzas["meat"] = 16.99
```

Have a look at the code. It should look like this:

The screenshot shows a Xcode playground window titled "Ready | Today at 12:35 PM". The code in the playground is as follows:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ["veggie": 14.99, "meat": 16.99])
/*(key "meat", value 16.99)
/*(key "veggie", value 14.99)

var dictPizzas = ["veggie": 14.99]
/*(key "veggie", value 14.99)

dictPizzas["meat"] = 17.99
17.999999999999...
dictPizzas["meat"] = 16.99
16.989999999999...
```

The output pane shows the state of the dictionary after each modification:

- After `dictPizzas["meat"] = 17.99`:
17.99
- After `dictPizzas["meat"] = 16.99`:
16.99

Instead of using the shorthand syntax, you can use the `updateValue(_:_forKey:)` method. This method does almost the same thing as the shorthand syntax. If the value does not exist, it creates the item; if it does exist, it will update the item. The only difference is that, when using the `updateValue(_:_forKey:)` variable, it actually returns the old value after performing the update. Using this method, you will get an optional value because it's possible that no value exists in the dictionary. Now, let's change the value from 16.99 to 15.99:

```
if let oldValue = dictPizzas.updateValue(15.99, forKey: "meat") {
    print("old value \(oldValue)")
}
```

Your code should now look like this:

The screenshot shows a Xcode editor window with a Swift script named '0011-Programming-for-Beginners-Ch5'. The code demonstrates various dictionary operations:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ("veggie", 14.99), ("meat", 16.99))
  ▶(key "meat", value 16.99)
  ▶(key "veggie", value 14.99)

var dictPizzas = ["veggie": 14.99]
  ▶(key "veggie", value 14.99)

dictPizzas["meat"] = 17.99
  17.999999999999...
dictPizzas["meat"] = 16.99
  16.999999999999...

if let oldValue = dictPizzas.updateValue(15.99, forKey: "meat") {
    print("old value \(oldValue)")
}
old value 16.99
```

Since we do not need the old value, we will just use the shorthand syntax to add a couple more pizzas:

```
dictPizzas["specialty"] = 18.99
dictPizzas["chicken"] = 16.99
```

Your code and output should now look like this:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ("veggie", 14.99), ("meat", 16.99))
    ▶ (key "meat", value 16.99)
    ▶ (key "veggie", value 14.99)

var dictPizzas = ["veggie": 14.99]
    ▶ (key "veggie", value 14.99)

dictPizzas["meat"] = 17.99
17.999999999999...
dictPizzas["meat"] = 16.99
16.999999999999...
dictPizzas["specialty"] = 18.99
18.999999999999...
dictPizzas["chicken"] = 16.99
16.999999999999...

if let oldValue = dictPizzas.updateValue(15.99, forKey: "meat") {
    print("old value \(oldValue)")
}

old value 16.99
```

Now that we have some data inside our dictionary, let's see how we can access that data.

Accessing an item in a dictionary

When trying to access an item inside a dictionary, you will always receive an optional value. The reason for this is that you could potentially receive `nil` if the value does not exist. So, you should always use an `if...let` statement to safeguard your code:

```
if let numChickenPrice = dictPizzas["chicken"] {
    print(numChickenPrice)
}
```

Your code should now look like this:

The screenshot shows an Xcode Playgrounds window titled "Ready | Today at 12:38 PM". The code in the playground is as follows:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ("veggie", 14.99), ("meat", 16.99))
var dictPizzas = ["veggie": 14.99]
dictPizzas["meat"] = 17.99
dictPizzas["meat"] = 16.99

if let oldValue = dictPizzas.updateValue(15.99, forKey: "meat") {
    print("old value \(oldValue)")
}

dictPizzas["specialty"] = 18.99
dictPizzas["chicken"] = 16.99

if let numChickenPrice = dictPizzas["chicken"] {
    print(numChickenPrice)
}
```

The right side of the screen shows the state of the dictionary after each operation:

- After `dictPizzas["meat"] = 17.99`:
 - "meat": 16.99
 - "veggie": 14.99
 - 17.99
 - 16.99
- After `dictPizzas["meat"] = 16.99`:
 - 18.99
 - 16.99
- After `print("old value \(oldValue)")`:
 - "old value 16.99"
- After `dictPizzas["chicken"] = 16.99`:
 - "16.99"

At the bottom of the playground, the output is shown in a scrollable list:
old value 16.99
16.99

Iterating over dictionary values

Just like an array, we can iterate through our dictionary. However, there are a few differences. Since a dictionary is unordered, each time you loop through, the values will never be in the same order. With dictionaries, you can loop through both the values and keys.

Let's iterate over a dictionary's values using a `for...in` loop. Add the following to Playgrounds:

```
for value in dictPizzas.values {
    print(value)
}
```

Your code should now look like this:

The screenshot shows an Xcode Playgrounds window titled "Ready | Today at 12:38 PM". The playground code is as follows:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: {"veggie": 14.99, "meat": 16.99})
var dictPizzas = ["veggie": 14.99]
dictPizzas["meat"] = 17.99
dictPizzas["meat"] = 16.99

if let oldValue = dictPizzas.updateValue(15.99, forKey: "meat") {
    print("old value \(oldValue)")
}

dictPizzas["specialty"] = 18.99
dictPizzas["chicken"] = 16.99

if let numChickenPrice = dictPizzas["chicken"] {
    print(numChickenPrice)
}

for value in dictPizzas.values {
    print(value)
}
```

The playground output pane shows the results of the code execution:

```
old value 16.99
16.99
17.99
16.99
14.99
18.99
```

The right side of the window displays the state of the variables:

- dictFirstExample: []
- dictSecondExample: []
- dictThirdExample: [{"meat": 16.99, "ve...": 14.99}]
- dictPizzas: [{"veggie": 14.99, "meat": 17.99, "meat": 16.99, "specialty": 18.99, "chicken": 16.99}]

Iterating over dictionary keys

To iterate over a dictionary's keys using a `for...in` loop, add the following to Playgrounds:

```
for key in dictPizzas.keys {
    print(key)
}
```

Your code and output should now look like this:

The screenshot shows an Xcode project window titled "ICG11-Programming-for-Beginners-Ch5". The code in the editor is as follows:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ("veggie", 14.99), ("meat", 16.99))
var dictPizzas = ["veggie": 14.99]
dictPizzas["meat"] = 17.99
dictPizzas["meat"] = 16.99

if let oldValue = dictPizzas.updateValue(15.99, forKey: "meat") {
    print("old value \(oldValue)")
}

dictPizzas["specialty"] = 18.99
dictPizzas["chicken"] = 16.99

if let numChickenPrice = dictPizzas["chicken"] {
    print(numChickenPrice)
}

for value in dictPizzas.values {
    print(value)
}

for value in dictPizzas.keys {
    print(value)
}
```

The output pane shows the results of the printed statements:

```
old value 16.99
16.99
16.99
18.99
14.99
18.99
chicken
meat
veggie
specialty
-----
meat
veggie
specialty
```

Iterating over dictionary keys and values

When you need to iterate over both dictionary keys and values using a `for...in` loop, you use the following:

```
for (key, value) in dictPizzas {
    print("\(key): \(value)")
}
```

Your code and output should now look like this:

The screenshot shows an Xcode project window titled "IO611-Programming-for-Beginners-Ch5". The code in the editor is as follows:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ("veggie", 14.99), ("meat", 16.99))
var dictPizzas = ["veggie": 14.99]
dictPizzas["meat"] = 17.99
dictPizzas["meat"] = 16.99

if let oldValue = dictPizzas.updateValue(15.99, forKey: "meat") {
    print("old value \(oldValue)")
}

dictPizzas["specialty"] = 18.99
dictPizzas["chicken"] = 16.99

if let numChickenPrice = dictPizzas["chicken"] {
    print(numChickenPrice)
}

for value in dictPizzas.values {
    print(value)
}

for value in dictPizzas.keys {
    print(value)
}

for (key, value) in dictPizzas {
    print("\(key): \(value)")
}
```

The output pane shows the results of the print statements:

```
old value 16.99
16.99
16.99
15.99
14.99
18.99
chicken
meat
veggie
specialty
chicken: 16.99
meat: 15.99
veggie: 14.99
specialty: 18.99
```

We have successfully looked at how to loop through a dictionary.

Checking the number of items in a dictionary

In addition to keys and values, we have other useful properties. We can see the number of items in a dictionary using the `count` property. Let's try that by adding the following:

```
print("There are \(dictPizzas.count) total pizzas.")
```

Now, your code and output should look like this:

The screenshot shows a Xcode playground window titled "ios11-Programming-for-Beginners-Ch5". The code in the playground is as follows:

```
let dictFirstExample = Dictionary<String, String>()
let dictSecondExample = [String: Int]()

var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral: ("veggie", 14.99), ("meat", 16.99))
var dictPizzas = ["veggie": 14.99]
dictPizzas["meat"] = 17.99
dictPizzas["meat"] = 16.99

if let oldValue = dictPizzas.updateValue(15.99, forKey: "meat") {
    print("old value \(oldValue)")
}

dictPizzas["specialty"] = 18.99
dictPizzas["chicken"] = 16.99

if let numChickenPrice = dictPizzas["chicken"] {
    print(numChickenPrice)
}

for value in dictPizzas.values {
    print(value)
}

for value in dictPizzas.keys {
    print(value)
}

for (key, value) in dictPizzas {
    print("\(key): \(value)")
}

print("There are \(dictPizzas.count) total pizzas.")
```

The right side of the screen shows the output of the code, which includes the state of the dictionaries and the printed values. The bottom pane shows the actual terminal output of the playground session.

```
old value 16.99
16.99
16.99
15.99
14.99
18.99
chicken
meat
veggie
specialty
chicken: 16.99
meat: 15.99
veggie: 14.99
specialty: 18.99
There are 4 total pizzas.
```

Along with a count, we can check whether a dictionary is empty. Let's use this in an `if...else` statement by adding the following:

```
if dictPizzas.isEmpty {
    print("there are no pizzas")
}
else {
    print("There are \(dictPizzas.count) total pizzas.")
}
```

Now, your code and output should look like this:

```
if dictPizzas.isEmpty {
    print("there are no pizzas")
} else {
    print("There are \(dictPizzas.count) total pizzas.")
}
```

There are 4 total pizzas.

"There are 4 total pi..."

This kind of logic is helpful when you want to display something back to the user or hide a UI.

Removing items from a dictionary

Next, let's learn how to remove an item from a dictionary. When deleting items from a dictionary, we have two primary ways of doing this. The first uses `removeValue(forKey:)`. Let's add this right above our `if...else` statement that checks whether the dictionary is empty:

```
dictPizzas.removeValue(forKey: "chicken")
```

Your code should now look like this:

```
dictPizzas.removeValue(forKey: "chicken")

if dictPizzas.isEmpty {
    print("there are no pizzas")
} else {
    print("There are \(dictPizzas.count) total pizzas.")
}
```

There are 3 total pizzas.

"There are 3 total pi..."

Let's look at the second way of removing dictionary items, which is by using the shorthand syntax. Add the following to Playgrounds, following on from the `removeValue(forKey:)` variable:

```
dictPizzas["meat"] = nil
```

Your code should now look like this:



The screenshot shows the Xcode Playgrounds interface with the title bar "Running iOS11-Programming-for-Beginners-Ch5". In the code editor, there is a script named "iOS11-Programming-for-Beginners-Ch5". The code contains the following:

```
dictPizzas.removeValue(forKey: "chicken")
dictPizzas["meat"] = nil

if dictPizzas.isEmpty {
    print("there are no pizzas")
} else {
    print("There are \(dictPizzas.count) total pizzas.")
}
```

The playground output window shows the results of the code execution:

```
16.99
nil
"There are 2 total pizzas."
```

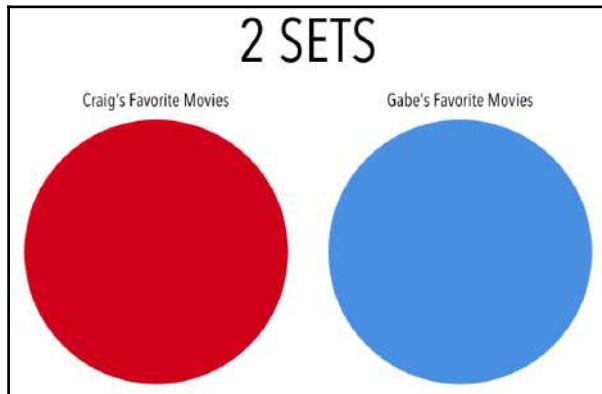
At the bottom of the playground window, the message "There are 2 total pizzas." is displayed.

Notice that, just like with `updateValue(_:_forKey:)`, `removeValue(forKey:)` will return you the value before it is removed. If you do not need the value, the shorthand syntax is the preferred method.

So far, we have covered arrays and dictionaries, and now we will review one last collection: sets.

Sets

A set stores unique values of the same type in a collection without a defined order. Let's look at the following diagram:



In the preceding diagram, we have two circles, both of which represent a set. On the left, we have Craig's favorite movies, and on the right, we have Gabe's favorite movies.

Creating an empty set

Before we create these sets, let's create an empty set and see what that looks like:

A screenshot of an Xcode interface. The window title is "iOS11-Programming-for-Beginners-Ch5". The status bar at the top right shows "Ready | Today at 1:19 PM". In the main editor area, there is a single line of code: "let movieSet = Set<String>()". To the right of the code, there is a small preview window showing "Set([])". The interface includes standard Xcode navigation and toolbars.

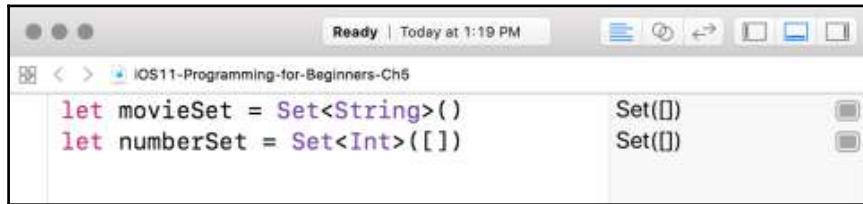
In this first set, after the equals sign, we create the set and give it a string data type. Then, we use the parentheses to initialize the set.

Creating a set with an array literal

Our first set was an empty string set, but we can create a set using an array literal. Let's add the following to Playgrounds:

```
let numberSet = Set<Int>([])
```

Your code should now look like this:



This preceding immutable set has an `int` data type, but, in the parentheses, we passed an empty array literal when we used the brackets.

Creating a mutable set

Now that we are familiar with the way sets are created, let's create a mutable set for Craig's favorite movies and one for Gabe's favorite movies. Add the following to Playgrounds:

```
var craigsFavMovieSet = Set<String>([])
var gabesFavMovieSet = Set<String>(["Fight Club", "Matrix", "Evil
Dead", "Big Trouble in Little China", "Aliens", "Winter Solider", "The
Illusionist", "Predator"])
```

Be aware that if you copy and past this code into Xcode, you might see a number of errors due to line breaks and book formatting.

Now, your code should look like this:

```
let movieSet = Set<String>()
let numberSet = Set<Int>([])

var craigsFavMovieSet = Set<String>([])
var gabesFavMovieSet = Set<String>(["Fight Club", "Matrix", "Evil Dead", "Big Trouble in Little China", "Aliens", "Winter Soldier", "The Illusionist", "Predator"])

0 "The Illusionist"
1 "Predator"
2 "Fight Club"
3 "Big Trouble in Little China"
4 "Winter Soldier"
5 "Matrix"
6 "Evil Dead"
7 "Aliens"
```

We now have two mutable sets. The first set is created with an empty array literal, and the second set is created with some initial values. Let's add some more items to both sets.

Adding items to a set

To add an item to a set, we have to use the `insert()` method. Let's use that to add another movie to Gabe's favorite movies:

```
gabesFavMovieSet.insert("Terminator")
gabesFavMovieSet
```

Your code should now look like this:

The screenshot shows an Xcode interface with a Swift file named `ViewController.swift` open. The code initializes two sets: `craigsFavMovieSet` and `gabesFavMovieSet`, both containing strings representing movie titles. The `gabesFavMovieSet` contains the titles from the previous list. A call to `gabesFavMovieSet.insert("Terminator")` is shown, along with the resulting output in the Results Panel: a set containing "The Illusionist", "Predator", and "Terminator".

```
let movieSet = Set<String>()
let numberSet = Set<Int>([])

var craigsFavMovieSet = Set<String>([])
var gabesFavMovieSet = Set<String>(["Fight Club", "Matrix", "Evil Dead", "Big Trouble in Little China", "Aliens", "Winter Soldier", "The Illusionist", "Predator"])

0 "The Illusionist"
1 "Predator"
2 "Fight Club"
3 "Big Trouble in Little China"
4 "Winter Soldier"
5 "Matrix"
6 "Evil Dead"
7 "Aliens"

gabesFavMovieSet.insert("Terminator")
Inserted true
memberAfterInsert! "Terminator"

gabesFavMovieSet
0 "The Illusionist"
1 "Predator"
2 "Terminator"
3 "Fight Club"
4 "Big Trouble in Little China"
5 "Winter Soldier"
6 "Matrix"
7 "Evil Dead"
8 "Aliens"
```

Now, Gabe has nine films, and Craig still has none. We added the `gabeFaveMovieSet` variable again so that we can see the contents update in the Results Panel. To add multiple items to a set, we can use an array literal.

Let's add ten films to Craig's list, as follows:

```
craigsFavMovieSet = ["The Pianist", "The Shawshank Redemption", "Dark Knight", "Black Swan", "Ip Man", "The Illusionist", "The Silence of the Lambs", "Winter Solider", "Green Mile", "Se7en"]
```

Your code should now look like this:



```
gabesFavMovieSet
0 "The Illusionist"
1 "Predator"
2 "Terminator"
3 "Fight Club"
4 "Big Trouble In Little China"
5 "Winter Soldier"
6 "Matrix"
7 "Evil Dead"
8 "Aliens"

craigsFavMovieSet = ["The Pianist", "The Shawshank Redemption", "Dark Knight", "Black Swan", "Ip Man", "The Illusionist", "The Silence of the Lambs", "Winter Soldier", "Green Mile", "Se7en"]
0 "Black Swan"
1 "Dark Knight"
2 "The Pianist"
3 "The Silence of the Lambs"
4 "The Shawshank Redemption"
5 "Se7en"
6 "Green Mile"
7 "The Illusionist"
8 "Winter Soldier"
9 "Ip Man"
```

Craig's set now has ten films. Next, let's look at how we can work with sets.

Checking whether a set contains an item

The first thing we can do with sets is check whether a set includes an item. Let's see whether Craig's movie list includes the movie `Green Mile`:

```
if craigsFavMovieSet.contains("Green Mile") {
    print("Green Mile found")
}
```

Your code should now look like this:



```
gabesFavMovieSet
0 "The Illusionist"
1 "Predator"
2 "Terminator"
3 "Fight Club"
4 "Big Trouble in Little China"
5 "Winter Soldier"
6 "Matrix"
7 "Evil Dead"
8 "Aliens"

craigsFavMovieSet = ["The Pianist", "The Shawshank Redemption", "Dark Knight", "Black Swan", "Ip Man", "The Illusionist", "The Silence of the Lambs", "Winter Solider", "Green Mile", "Se7en"]
0 "Black Swan"
1 "Dark Knight"
2 "The Pianist"
3 "The Silence of the Lambs"
4 "The Shawshank Redemption"
5 "Se7en"
6 "Green Mile"
7 "The Illusionist"
8 "Winter Soldier"
9 "Ip Man"

if craigsFavMovieSet.contains("Green Mile") {
    print("Green Mile found")
}

[Green Mile found]
```

In the preceding example, we used the `contains()` method to discover whether an item is in the set.

Iterating over a set

If we want a list of all the movies in Gabe's list, we can use a `for...loop`. Let's see how that works:

```
for movie in gabesFavMovieSet {
    print("Gabe's movie - \\" + movie + ")"
}
```

Your code should now look like this:

The screenshot shows an Xcode interface with a code editor and a playground output window. The code in the editor is as follows:

```
let movieSet = Set<String>()
let numberSet = Set<Int>([])

var craigsFavMovieSet = Set<String>([])
var gabesFavMovieSet = Set<String>(["Fight Club", "Matrix", "Evil Dead", "Big Trouble in Little China", "Aliens", "Winter Solider", "The Illusionist", "Predator"])
gabesFavMovieSet.insert("Terminator")
gabesFavMovieSet

craigsFavMovieSet = ["The Pianist", "The Shawshank Redemption", "Dark Knight", "Black Swan", "Ip Man", "The Illusionist", "The Silence of the Lambs", "Winter Solider", "Green Mile", "Se7en"]

if craigsFavMovieSet.contains("Green Mile") {
    print("Green Mile found!")
}

for movie in gabesFavMovieSet {
    print("Gabe's movie - \(movie)")
}
```

The playground output window shows the state of the sets and the results of the print statements:

```
Set() Set()
Set() Set()
Set()
("The Illusionist", "P...")
(inserted true, mem...)
("The Illusionist", "P...")
("Black Swan", "Dar...
"Green Mile found!" (9 times)

Gabe's movie - The Illusionist
Gabe's movie - Predator
Gabe's movie - Terminator
Gabe's movie - Fight Club
Gabe's movie - Big Trouble in Little China
Gabe's movie - Winter Solider
Gabe's movie - Matrix
Gabe's movie - Evil Dead
Gabe's movie - Aliens
```

Now that we have seen a `for...in` loop for all three collections, that is, arrays, dictionaries, and sets, you can see that there are a lot of similarities. Remember, since sets come unordered, every time we run our `for...in` loop, we will get a list in a different order. The way around this is to use the `sorted()` method. Using `sorted()` will ensure that every time we loop through our list, it will always be in the same order. Let's do that with Craig's movie list:

```
for movie in craigsFavMovieSet.sorted() {
    print("Craig's movie - \(movie)")
}
```

Your code should now look like this:

```
Ready | Today at 1:28 PM
```

```
Set<>()
```

```
Set<>()
```

```
Set<>()
```

```
{The Illusionist', 'P...}
```

```
Inserted true, mem...@
```

```
{'The Illusionist', 'P...}
```

```
{'Black Swan', 'Dar...}
```

```
"Green Mile found!"
```

```
(9 times)
```

```
(10 times)
```

```
Craig's movie - Black Swan
```

```
Craig's movie - Dark Knight
```

```
Craig's movie - Green Mile
```

```
Craig's movie - Ip Man
```

```
Craig's movie - Se7en
```

```
Craig's movie - The Illusionist
```

```
Craig's movie - The Pianist
```

```
Craig's movie - The Shawshank Redemption
```

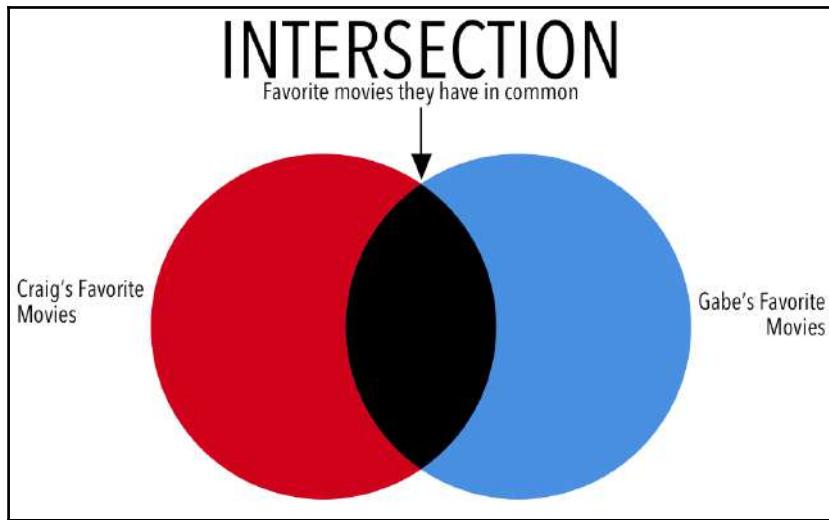
```
Craig's movie - The Silence of the Lambs
```

```
Craig's movie - Winter Solider
```

Now that we have our set sorted, let's look at the real power of using sets.

Intersecting two sets

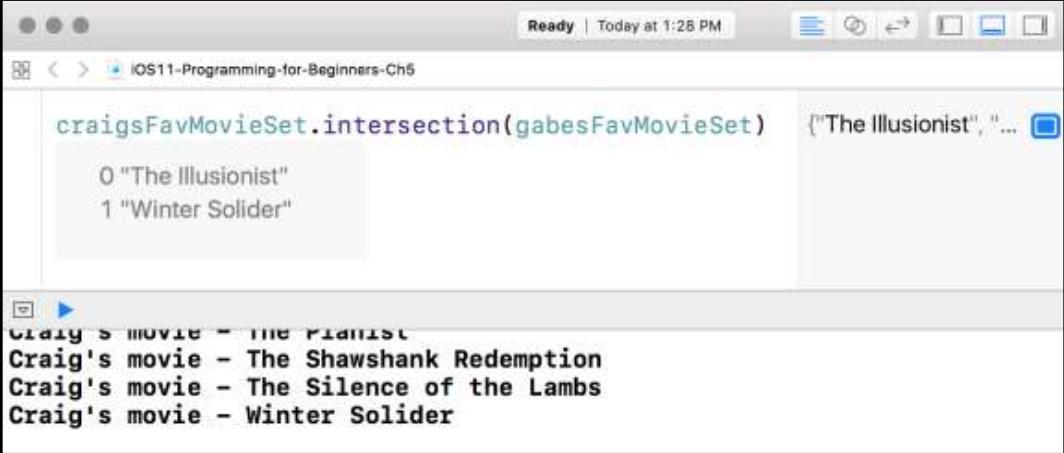
In the following diagram, we can see that, where both sets intersect, we should get a list of any movies they have in common:



We can do the same using the `intersection()` method in our code. Let's intersect both movie lists and see what happens:

```
craigsFavMovieSet.intersection(gabesFavMovieSet)
```

Your code and output should now look like this:



The screenshot shows an Xcode interface with a code editor and a terminal window.

In the code editor, the following code is shown:

```
craigsFavMovieSet.intersection(gabesFavMovieSet)
```

The output of the code is:

```
0 "The Illusionist"  
1 "Winter Solider"
```

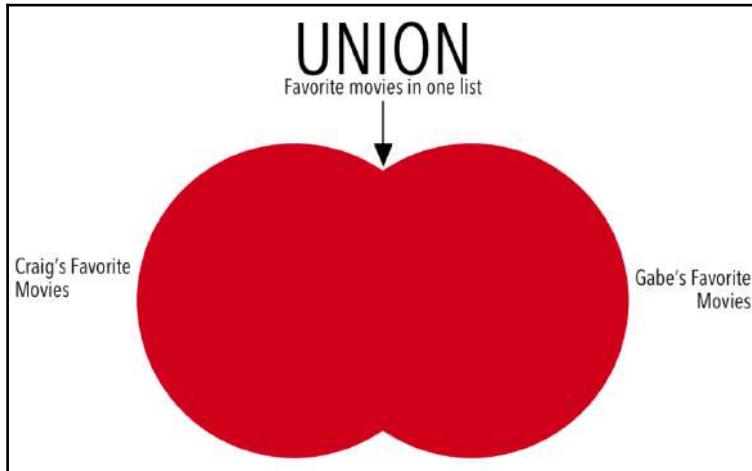
In the terminal window below, the output of the code is displayed:

```
Craig's movie - The Pianist  
Craig's movie - The Shawshank Redemption  
Craig's movie - The Silence of the Lambs  
Craig's movie - Winter Solider
```

We can see that the only two movies that these sets have in common are *The Illusionist* and *Winter Solider*. In addition to seeing which movies the two sets have in common, we can also join the lists to get one consolidated list of the movies from both sets.

Joining two sets

If you look at the following diagram, you will see the two sets joined together:



Using the `union()` method, we get a consolidated list of items with no duplicates. Let's try this in Playgrounds:

```
craigsFavMovieSet.union(gabesFavMovieSet)
```

Your code should now look like this:

```
craigsFavMovieSet.intersection(gabesFavMovieSet)
0 "The Illusionist"
1 "Winter Solider"

craigsFavMovieSet.union(gabesFavMovieSet)
0 "Black Swan"
1 "Dark Knight"
2 "The Pianist"
3 "The Shawshank Redemption"
4 "Big Trouble in Little China"
5 "Green Mile"
6 "Winter Solider"
7 "Matrix"
8 "Ip Man"
9 "Evil Dead"
10 "Predator"
11 "The Silence of the Lambs"
12 "Fight Club"
13 "Se7en"
14 "The Illusionist"
15 "Terminator"
16 "Aliens"
```

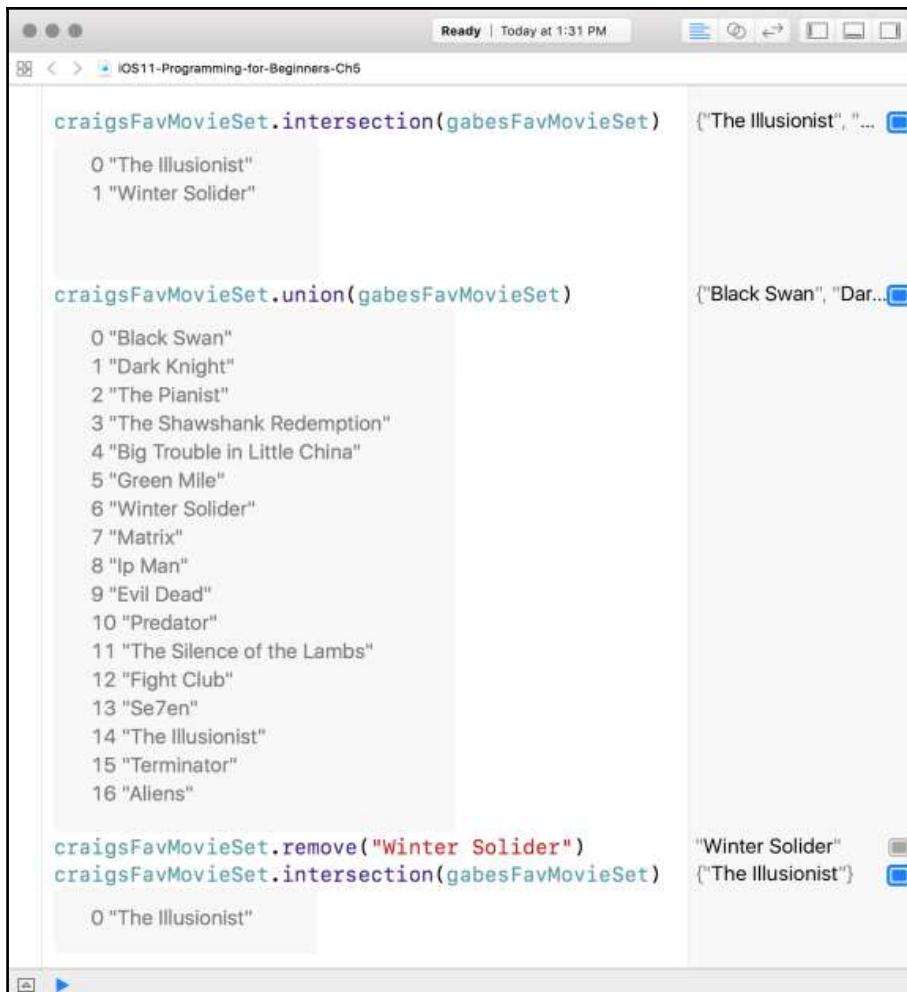
We have a combined list of movies that includes all of the movies that the two sets did not have in common, along with the two movies that were in common but listed only once. As you can see, sets are really powerful, and you can use them to manipulate data. Finally, we need to look at how we can remove items from a set.

Removing items from a set

To remove an item from a set, we can use the `remove()` method. When we use this method, we input the item we want to remove in the parentheses. Let's remove *Winter Solider* from Craig's movie list:

```
craigsFavMovieSet.remove("Winter Solider")
```

Your code should now look like this:



The screenshot shows the Xcode interface with a single output window displaying the results of running a script. The window title is "Ready | Today at 1:31 PM". The code in the window is as follows:

```
craigsFavMovieSet.intersection(gabesFavMovieSet) {"The Illusionist", ...}
0 "The Illusionist"
1 "Winter Solider"

craigsFavMovieSet.union(gabesFavMovieSet) {"Black Swan", "Dar...
0 "Black Swan"
1 "Dark Knight"
2 "The Pianist"
3 "The Shawshank Redemption"
4 "Big Trouble in Little China"
5 "Green Mile"
6 "Winter Solider"
7 "Matrix"
8 "Ip Man"
9 "Evil Dead"
10 "Predator"
11 "The Silence of the Lambs"
12 "Fight Club"
13 "Se7en"
14 "The Illusionist"
15 "Terminator"
16 "Aliens"

craigsFavMovieSet.remove("Winter Solider")
craigsFavMovieSet.intersection(gabesFavMovieSet) "Winter Solider"
0 "The Illusionist"
```

The output shows the initial intersection of two sets, which includes "The Illusionist" and "Winter Solider". The union of the sets is a list of 16 movies. After running the `remove("Winter Solider")` command, the intersection is shown again, but it no longer contains "Winter Solider", leaving only "The Illusionist".

If you tried to remove more than a single item from a set (for instance, all of the items), then you can use the `removeAll()` method or give it an empty array literal:

```
craigsFavMovieSet.removeAll()
gabesFavMovieSet = []
```

Your code should now look like this:



The screenshot shows an Xcode interface with a code editor and a debugger pane. The code editor contains the following code:

```
craigsFavMovieSet.intersection(gabesFavMovieSet)
craigsFavMovieSet.union(gabesFavMovieSet)
craigsFavMovieSet.remove("Winter Solider")
craigsFavMovieSet.intersection(gabesFavMovieSet)
craigsFavMovieSet.removeAll()

Set()
gabesFavMovieSet = []
Set()
```

The debugger pane shows the state of variables:

Variable	Type	Value
<code>craigsFavMovieSet</code>	Set	Set([])
<code>gabesFavMovieSet</code>	Set	Set([])

Now, both sets are empty.

Summary

We covered a lot in this chapter. We are now comfortable with using collections. Now that you are familiar with arrays, dictionaries, and sets, take the time to practice and work with them as much as you can. Collections are used a lot in programming, so getting comfortable is very important.

Even though we will touch on these things throughout the creation of the *Let's Eat* app, it is best to make sure that you are comfortable with what we covered here. So, please review the material as often as is necessary to make sure you feel that you are proficient in the topics contained in this chapter.

In the next chapter, we will start building our *Let's Eat* app. Over the next two chapters, we will work on getting our project set up, and then we will begin working on the visual aspects of our app.

6

Starting the UI Setup

Now that you have learned Swift, which will help you to understand a lot of the boilerplate code you will see later, it's time to start building our *Let's Eat* app. Let's begin by getting an overview of what we are going to build. We will review the finished product and then get into how to create this app. Before we start, there will be a lot of new terms and things with which you may or may not be familiar. Learn as much as you can, and do not let the finer details stop you from progressing.

We will cover the following topics in this chapter:

- Useful terms
- App tour
- Project setup
- Storyboards
- Creating a custom title view

Useful terms

Before we dig in and start getting our UI set up, we need to take a few minutes to introduce (or re-introduce) you to some terms that you should understand while we build our app:

- **View Controller**
- **Table View Controller**
- **Collection View Controller**
- **Navigation Controller**
- **Tab Bar Controller**
- **Storyboard**
- **Segue**
- **Stack Views**
- **Auto layout**
- **Model View Controller (MVC)**

View Controllers

View Controllers (`UIViewController`) are blank scenes that you can use to hold other UI elements. They give you the ability to create a custom interface.

Table View Controllers

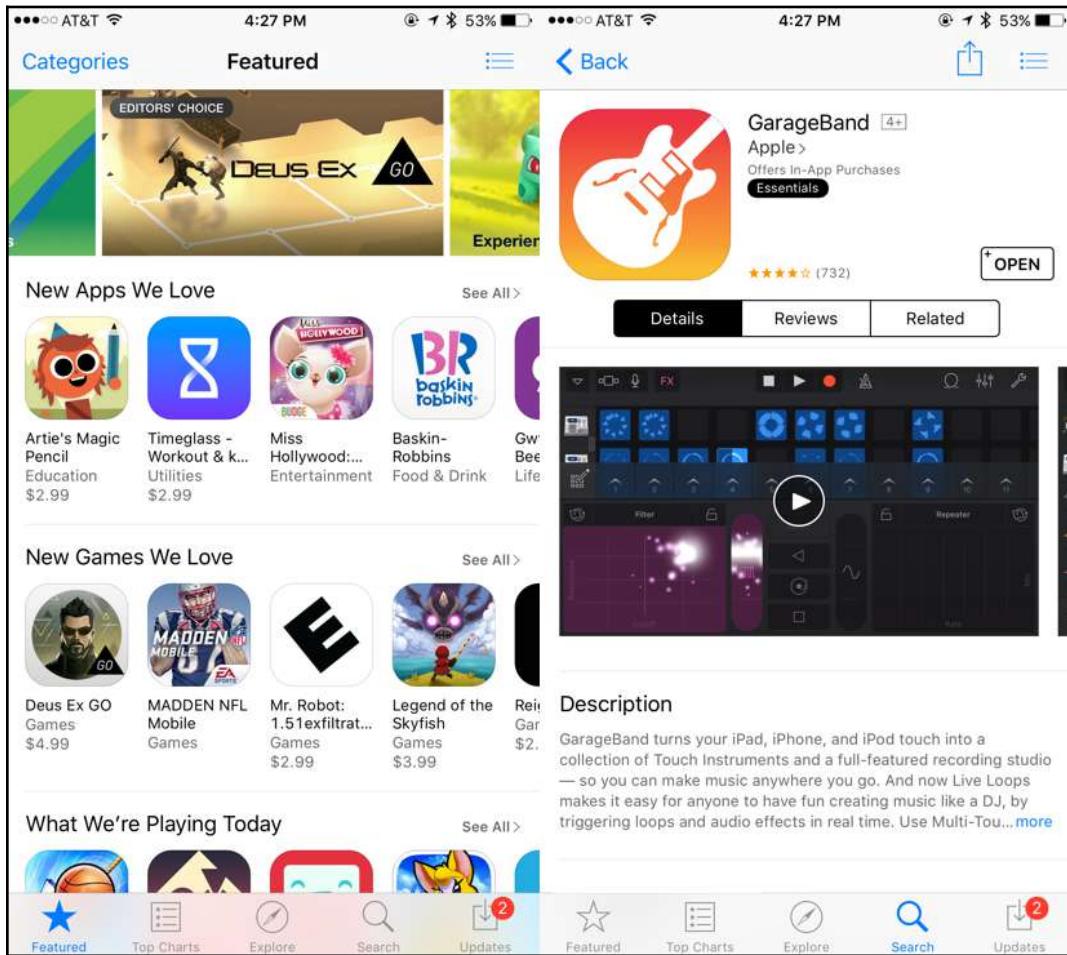
A Table View Controller (`UITableViewController`), which inherits from `UIViewController`, is one of the most common UI elements and is used to display a list of items. For example, Apple's **Settings** screen uses a Table View Controller to display the list of settings a user can access and change:



Collection View Controllers

Collection View Controllers (`UICollectionViewControllers`) are typically used when you want to display elements within a grid. They are highly customizable and, because of that, are becoming more popular in non-grid-based layouts.

The App Store, for example, currently uses `UICollectionViewControllers` for both its featured page and its app details page:



Navigation Controllers

A Navigation Controller (`UINavigationController`) is a UI element that allows you to build a drill-down interface for hierarchical content. When you embed a View Controller, Table View Controller, or Collection View Controller into a Navigation Controller, it manages navigation from one controller to another.

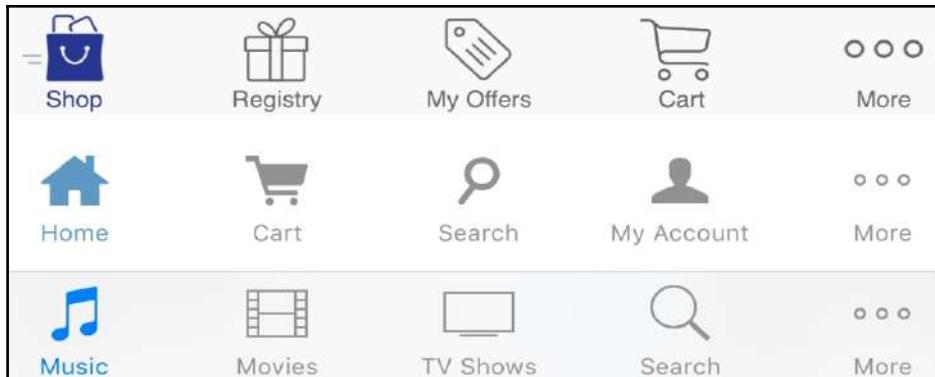
Tab Bar Controllers

The Tab Bar Controller (`UITabBarController`) manages an array of View Controllers. The *Let's Eat* app will use a Tab Bar Controller. This controller will give us the ability to have navigation for our app with minimal setup.

Apple has a few apps with which you might be familiar that use the Tab Bar Controller:

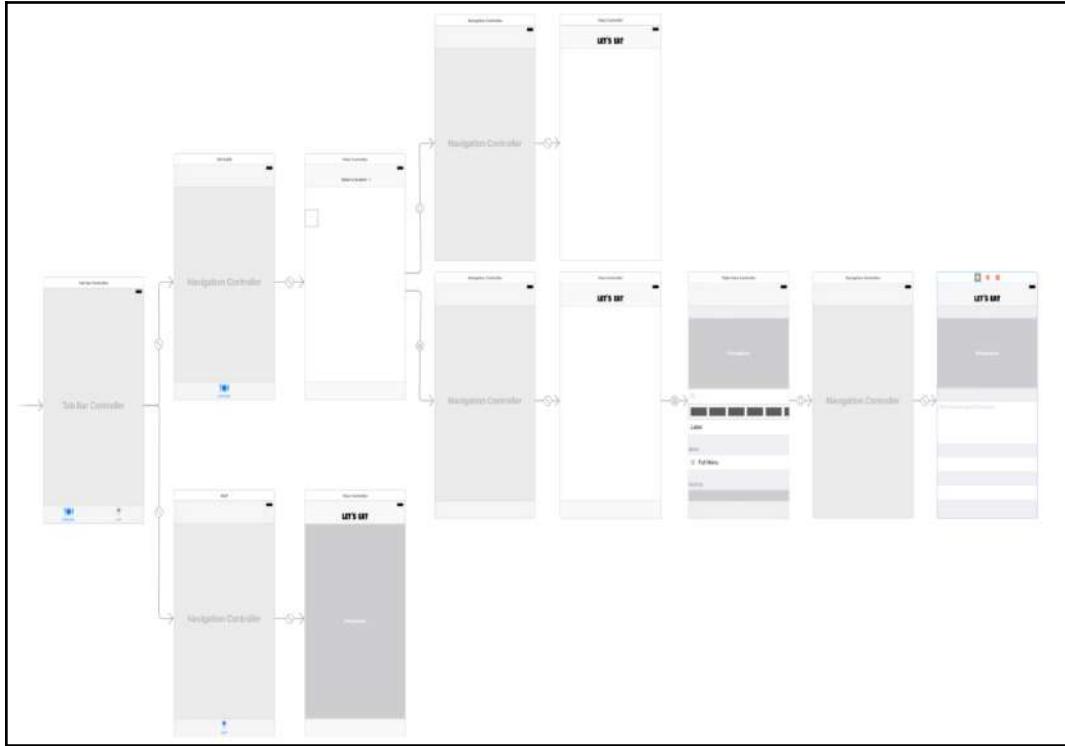


`UITabBarController` can only have five tabs on the iPhone. If your `UITabBarController` has more than five tabs on the iPhone, the fifth tab, and any after that, move underneath a **More** button:



Storyboards

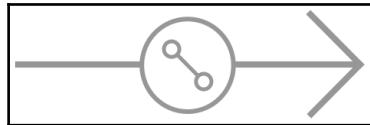
A storyboard is a file that displays a visual representation of your app's UI. The following is what a storyboard looks like for an app:



Storyboards let you create your entire app visually using View Controllers, Table View Controllers, and Collection View Controllers as scenes. Along with building your app visually, you can connect scenes and set up transitions between scenes using segues.

Segues

Segues are used to connect one controller to another. In the storyboard, segues are represented by an arrow with an icon:



Segues also give you the ability to specify a transition from one scene to another, with very little to no programming.

Stack Views

Stack Views are a great way to stack different components either horizontally or vertically. We will cover Stack Views in this book because they are a great way to easily organize components with equal spacing.

Auto Layout

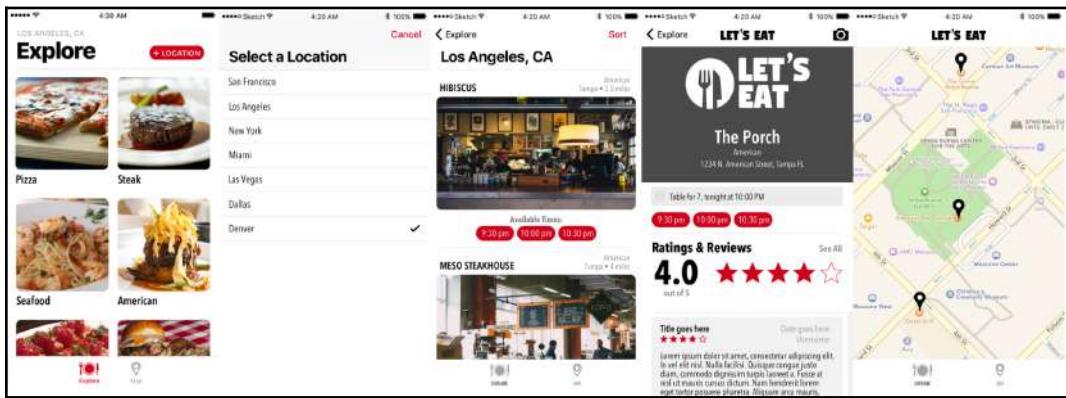
Auto layout is an excellent tool that allows you to support different screen sizes and device-rotation. With auto layout, you can set different constraints on UI elements for it to adjust to changes in size and rotation. Using auto layout in your app allows you to use one storyboard for all devices.

Model View Controller (MVC)

MVC is a standard software design pattern, which is a solution for commonly occurring problems within software design. Apple has built iOS apps on the MVC design pattern. This pattern divides our app into three camps, known as the Model, View, and Controller. We will cover this in detail later in this book.

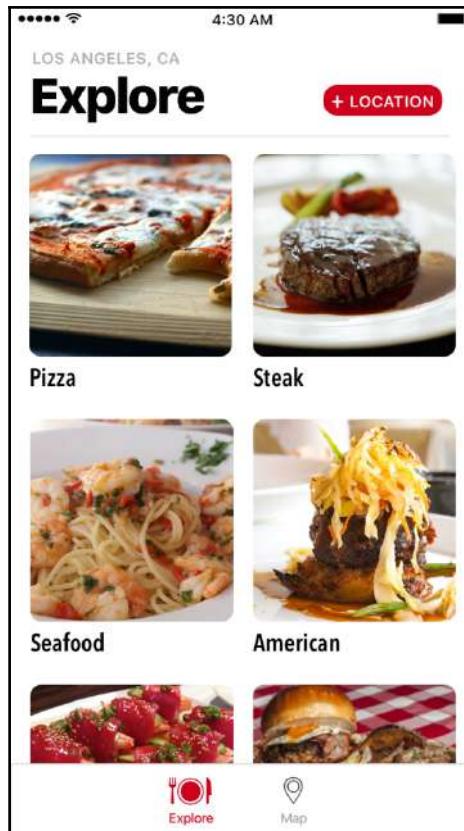
App tour

The *Let's Eat* app that we are building is a restaurant reservation app that allows users to find restaurants in a specific area and create reservations from within the app (although our app does not book those reservations). I chose a restaurant reservation app for the lessons in this book because most of the new iOS 12 features work well together in such an app. The app covers a lot of different aspects, from maps to iMessage extensions. Let's take a look at the overall flow of the app, so that, as we build, you have a good idea of the direction we are heading in:



The Explore tab

When the app launches, you will see the **Explore** tab. This tab allows users to search for a particular cuisine and to set their location. Let's break down each component in this view:

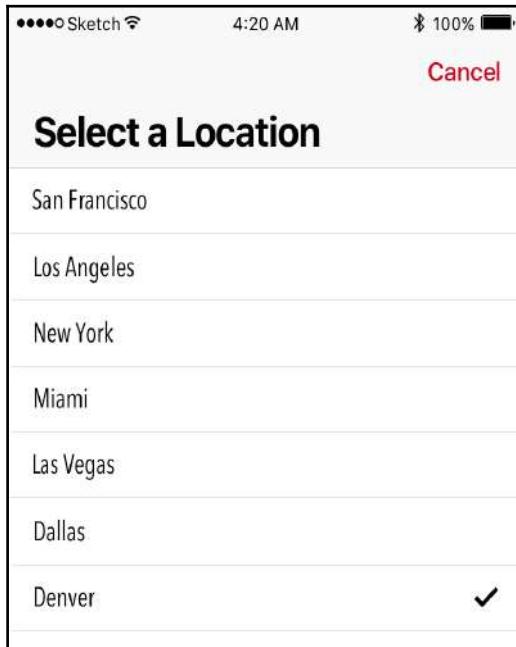


For this screen, we will work with an empty View Controller, which is where all of our UI components live. As you can see, this view in our app is designed to be a grid so that we will be using a Collection View Controller. We will be setting up this Collection View Controller ourselves.

When I build apps, I typically start with a blank Collection View or Table View, because it gives me more flexibility in my code as well as with my user interface.

Locations

The Locations view is a list of cities accessed from the **Explore** tab. We load a list of cities from a local file and, when the user selects a city, the app loads all of the restaurants from that area:



For this Locations view, we will be working with a View Controller that uses a Table View.

Restaurant listings

In Restaurant listings, we can see restaurants in the area by the selected cuisine:

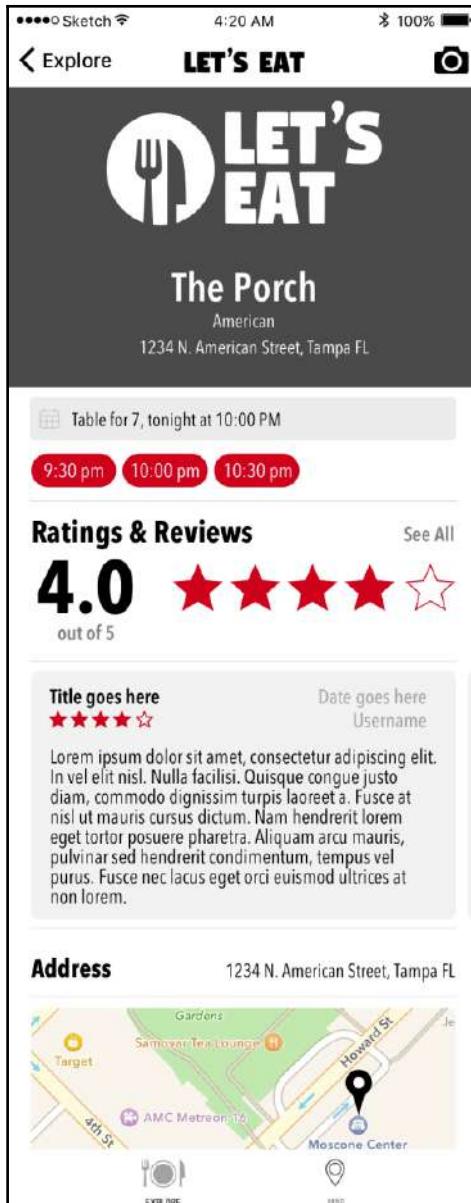


We will be covering both `UICollectionViews` and `UITableViews` in this book, but, as an introduction, you should know that `UICollectionViews` are very powerful—this is because you can customize them to look how you want. For example, the App Store detail is a custom `UICollectionView`.

One great feature when using `UICollectionView` is that, when you are building a universal app such as this one, you can make your view look like a list for the iPhone, but appear as a grid on the iPad with minimal effort.

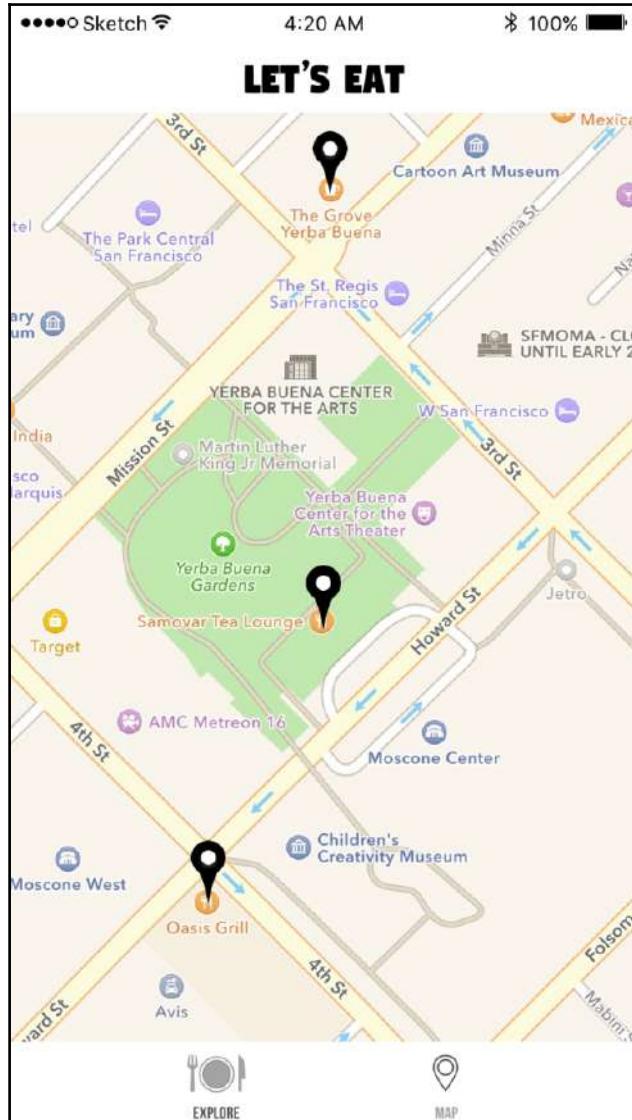
Restaurant detail

Our Restaurant detail has more information about the restaurant. This view is built using a UITableView that uses static cells:



The Map tab

Our **Map** tab is a View Controller with a map that has pins dropped on it from a specific location, denoting all of the restaurants in the area:



Project setup

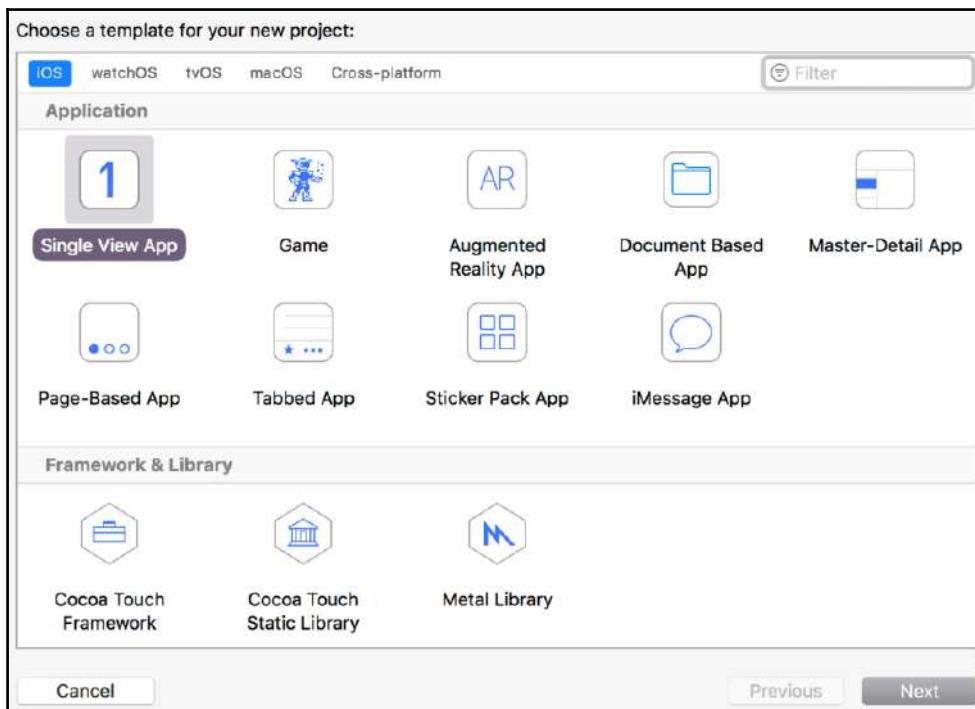
Now that we have gotten a tour of the app, we are going to build the *Let's Eat* app. First, we need to create the app, then work on the UI and, lastly, design our app in a storyboard.

For the initial setup of the app, we will look at some basics of iOS, starting with creating a new project.

Creating a new project

To create a new project, do the following:

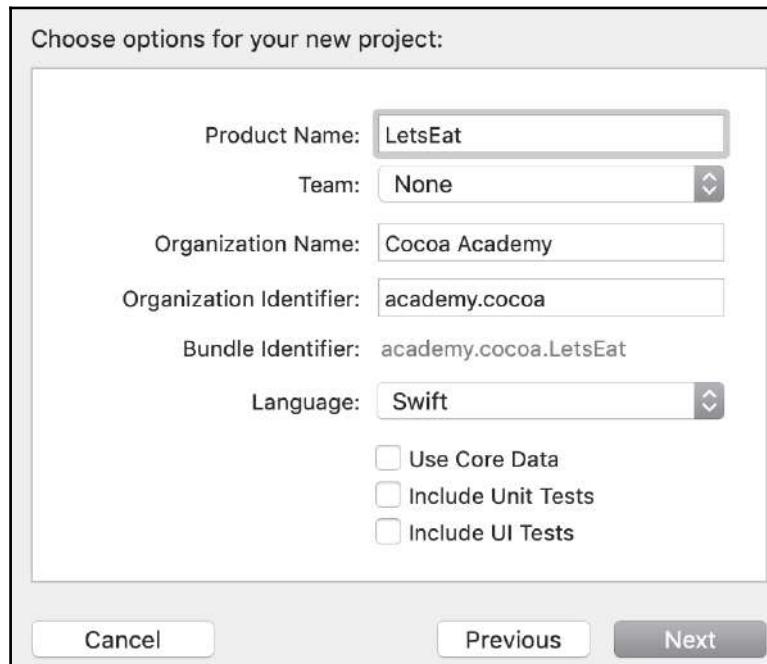
1. Open Xcode and the Xcode welcome screen will appear. Click on **Create a new Xcode project** in the left panel of the welcome screen.
2. Select **Single View App** and click on **Next**:



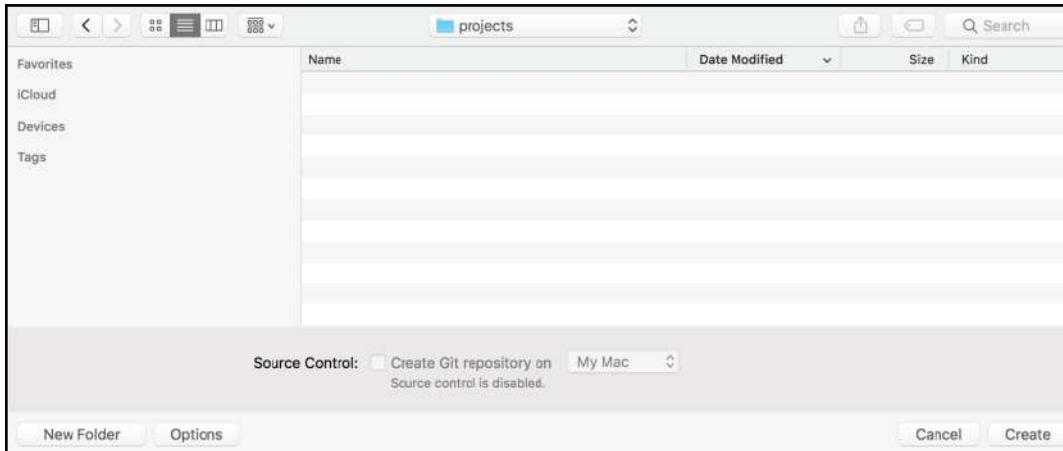
3. In the options screen that will appear, there will be a number of items to complete or choose. Add the following into that options screen and then hit **Next**:

- **Product Name:** LetsEat
- **Team:** Your account, or leave blank
- **Organization Name:** Your name/company name
- **Organization Identifier:** Your domain name in reverse order
- **Language:** Swift
- **Use Core Data:** Unchecked
- **Include Unit Tests:** Unchecked
- **Include UI Tests:** Unchecked

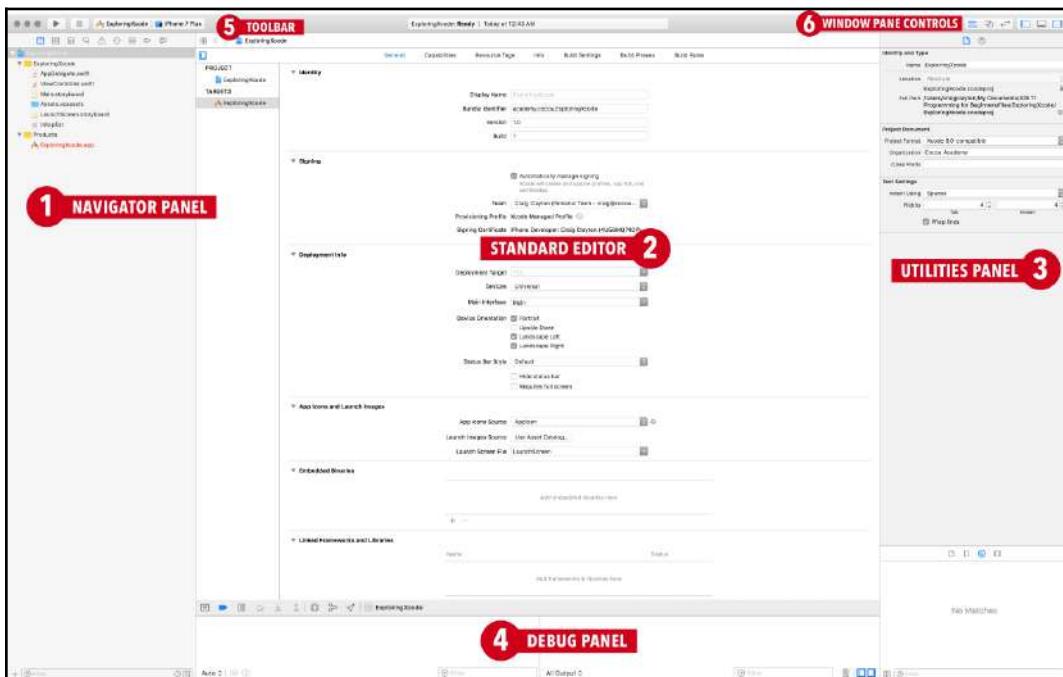
Your screen should look like the following screenshot:



4. Choose where to save your project, and then hit **Create**:



5. You're presented with the following screen:



Your project is created, and we can start working on building our first iOS app.

Summary

In this chapter, we covered useful terms that we will use throughout this book. We also looked at what we are going to build within the app, and now we have a good idea of what the app will look like when we are done.

Next, we'll start working inside the storyboard and getting the UI of the application set up. Once we have everything set up, we will focus on code throughout the rest of the book. If you are familiar with working with storyboards or do not want to learn the design aspect of iOS, please skip to Chapter 11, *Getting Started with the Grid*.

7

Setting Up the Basic Structure

Typically, before I write any code when working on a project, I like to set up my storyboard, which allows me to focus on coding without having to go back and forth from storyboard to code. In this book, we will do some of our layouts in code to show you how to do that. But first, as I mentioned earlier, my preference is to set up as much as I can inside of the storyboard.

The following will be covered in this chapter:

- Creating a Tab Bar Controller
- Tab Bar buttons
- Launch screens
- Navigation Controllers

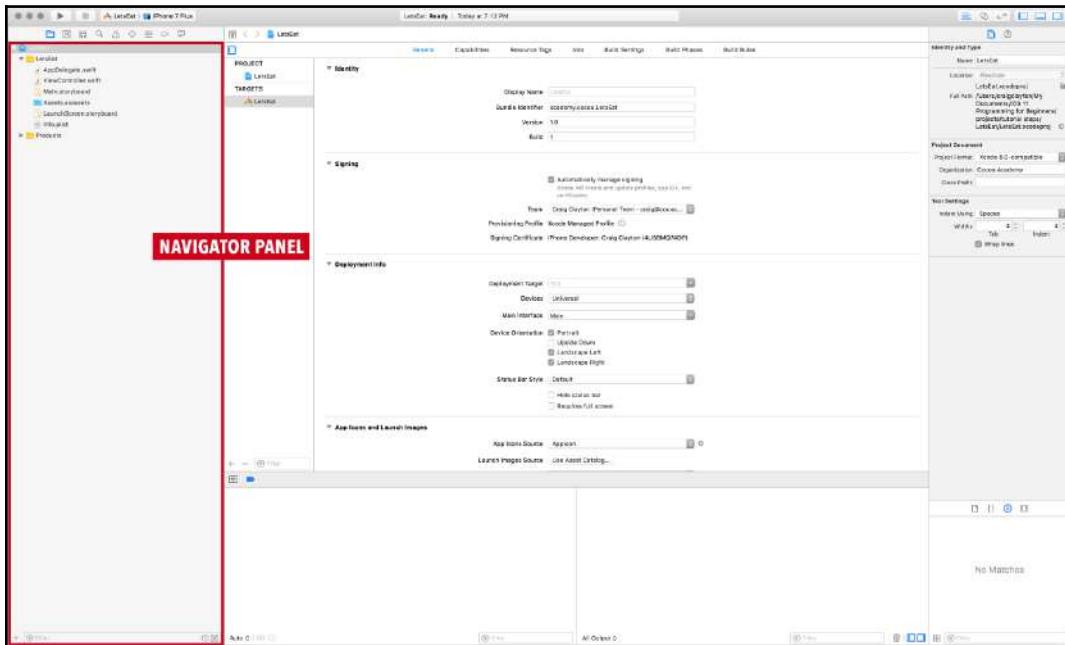
In the last chapter, we created our project, and now we are going to continue with that by building a Tab bar Controller from scratch. Although there is a Tab Bar Controller template that has everything you need, I find that starting from scratch is an excellent way to learn. Also, I find that it is easier to start to clean, rather than fix or update, the template. However, you may want to utilize the template to begin your project in the future. Let's start setting up our app.

Starting from scratch

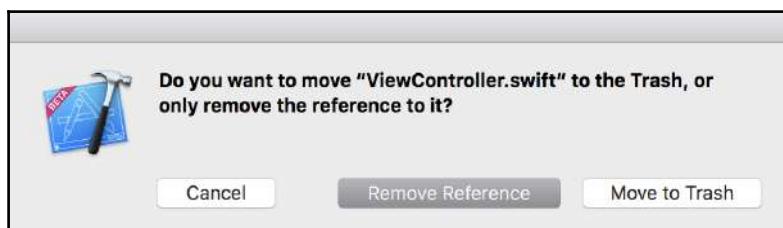
We will be creating all of our files from scratch, so we will delete the existing files in our project and recreate them in the coming chapters. The reason for this is so you can become comfortable with a project and understand how it was set up.

To delete the ViewController.swift file, do the following:

1. Select the ViewController.swift file in the NAVIGATOR PANEL:



2. With the file selected, hit the *Delete* or *Backspace* key. You will get the following message:



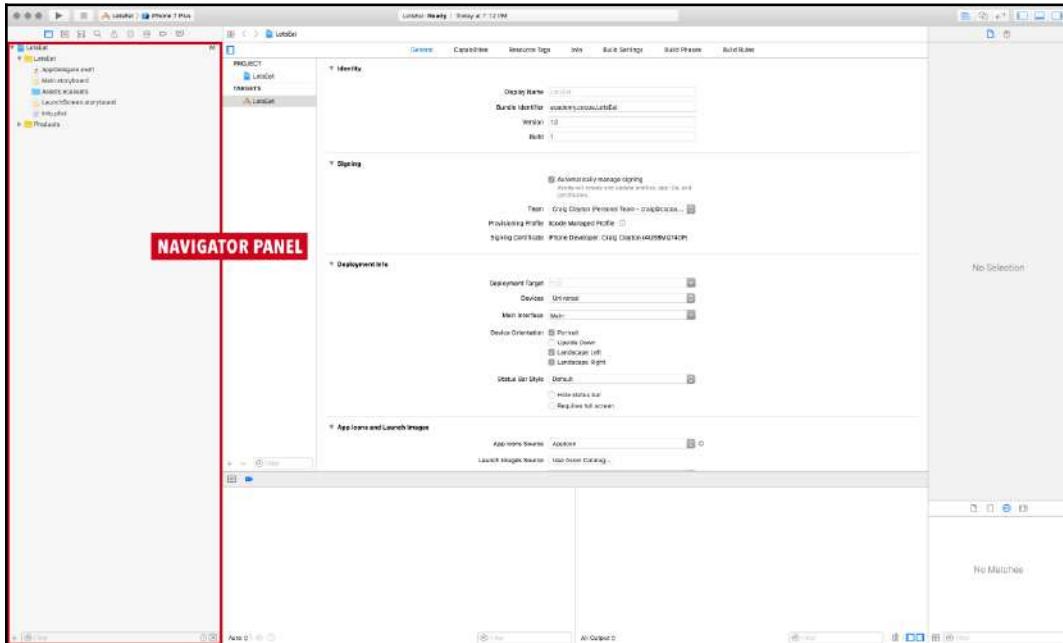
3. Select **Move to Trash**.

Now, we can continue with the setup of the storyboard.

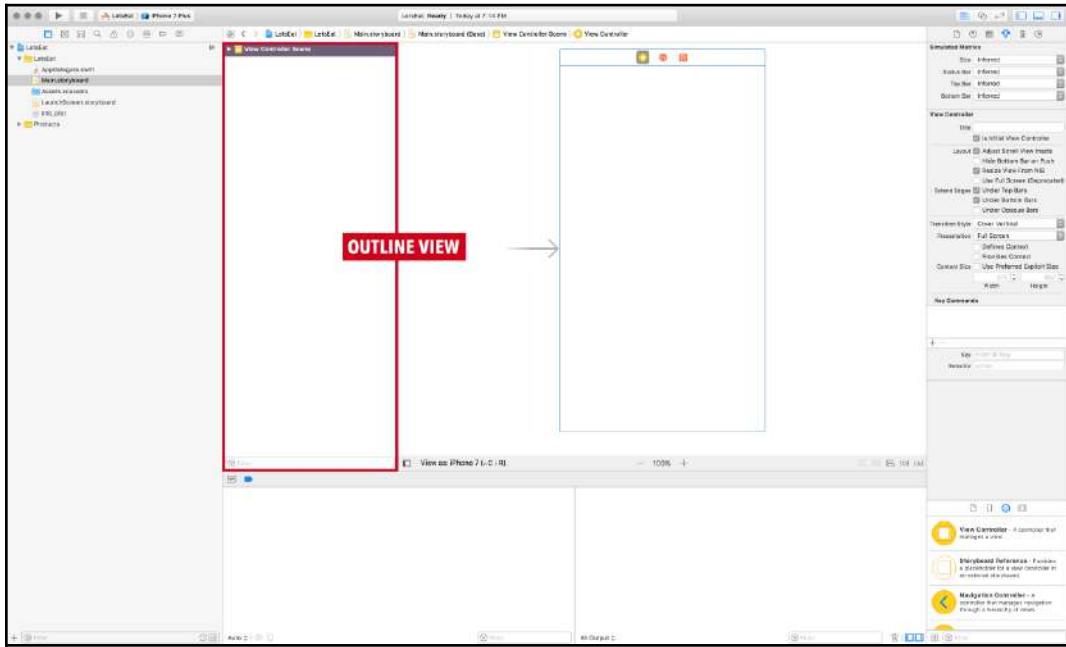
Storyboard setup

Let's get familiar with the UI setup. To update your `Main.storyboard`, do the following:

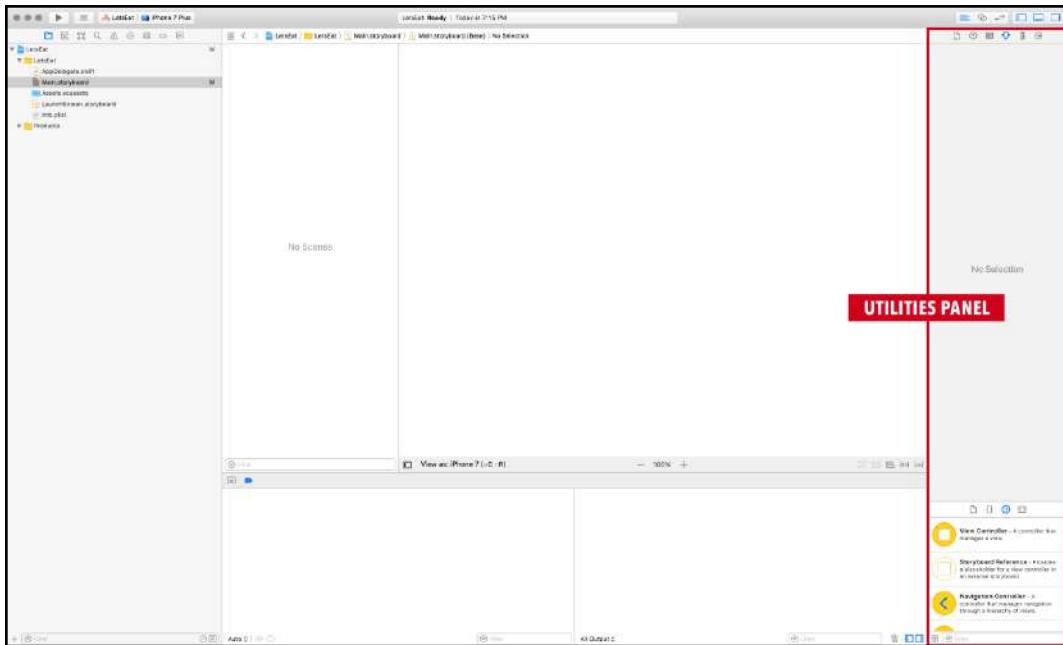
1. Select the `Main.storyboard` file in the Navigator panel:



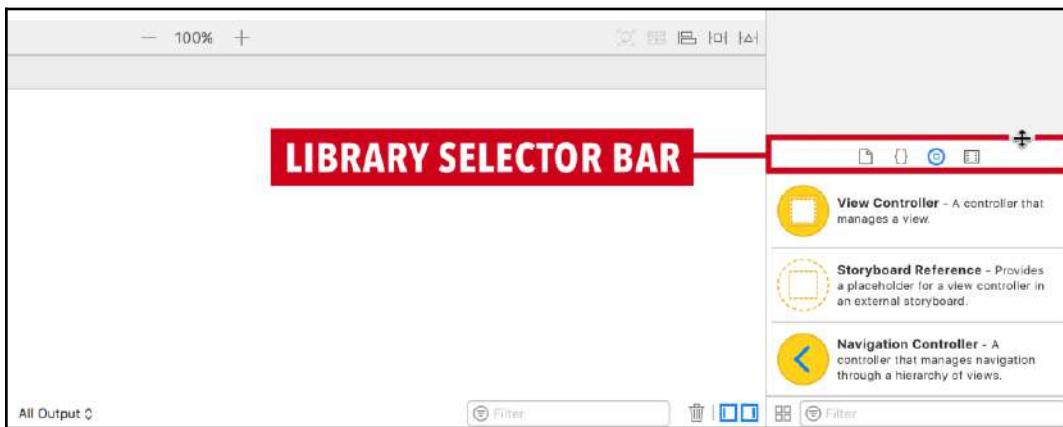
2. In this storyboard file, select **View Controller scene** in the **OUTLINE VIEW**:



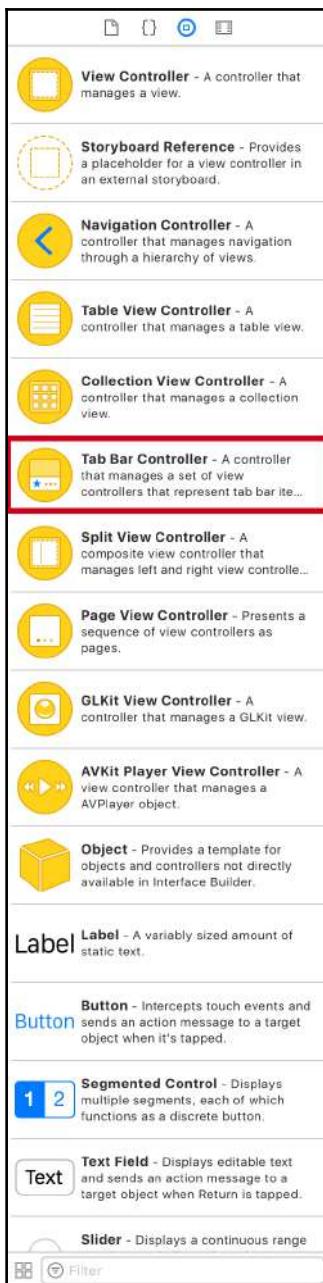
3. With the scene selected, press the *Delete* or *Backspace* key, and now your **Main.storyboard** file will be empty.
4. In your **UTILITIES PANEL**, in the bottom pane, you will see the **LIBRARY SELECTOR BAR**. In the bar, select the object library:



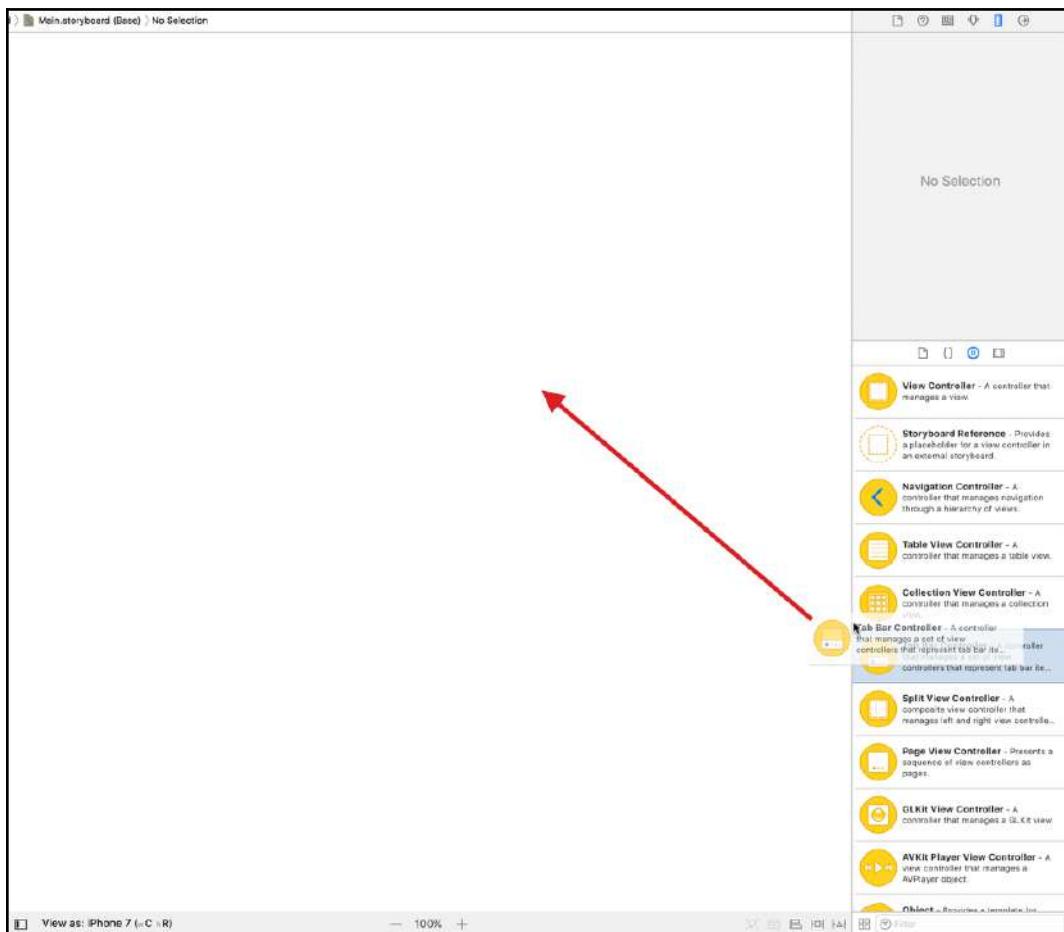
5. Pull up on the **LIBRARY SELECTOR BAR** to view more of the object library:



6. Find the Tab Bar Controller:



7. Drag the Tab Bar Controller out onto the canvas:



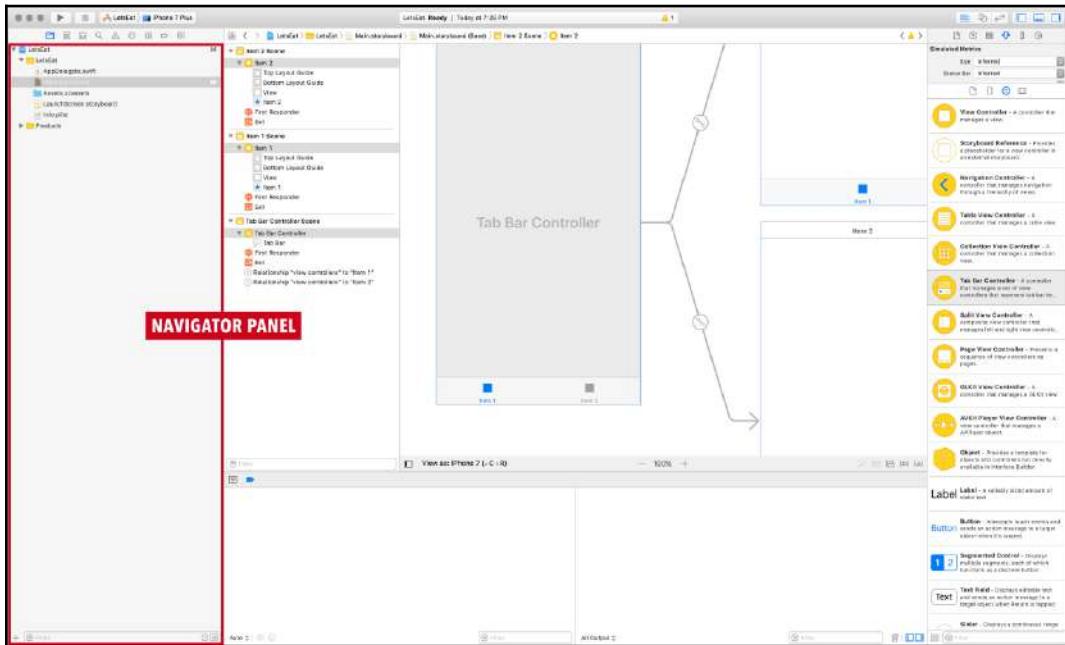
We now have our **Tab Bar Controller**, which will only have two tabs.

Next, we will get our app assets set up so that we can give our tabs image icons.

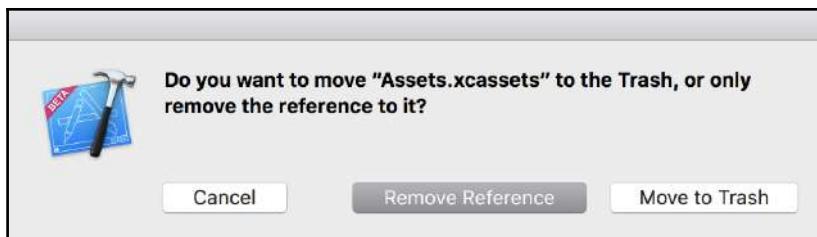
Adding our app assets

Let's add images into our project by performing the following:

1. Select the Assets.xcassets folder in the NAVIGATOR PANEL:

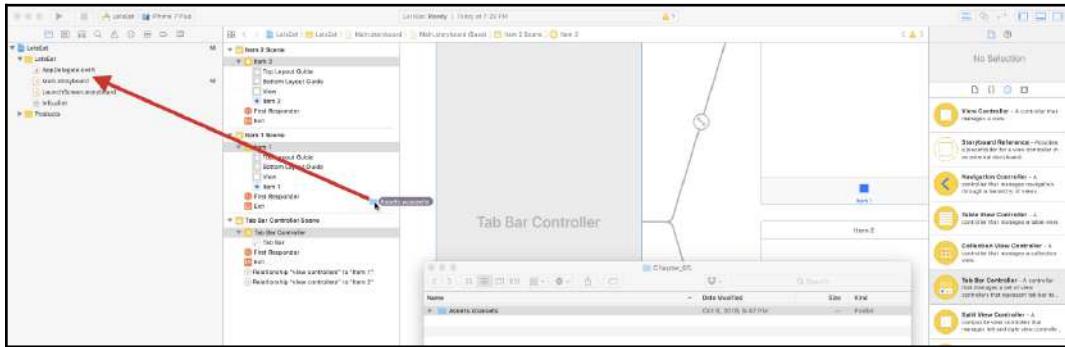


2. Hit the *Delete* or *Backspace* button, and you will get the following message:

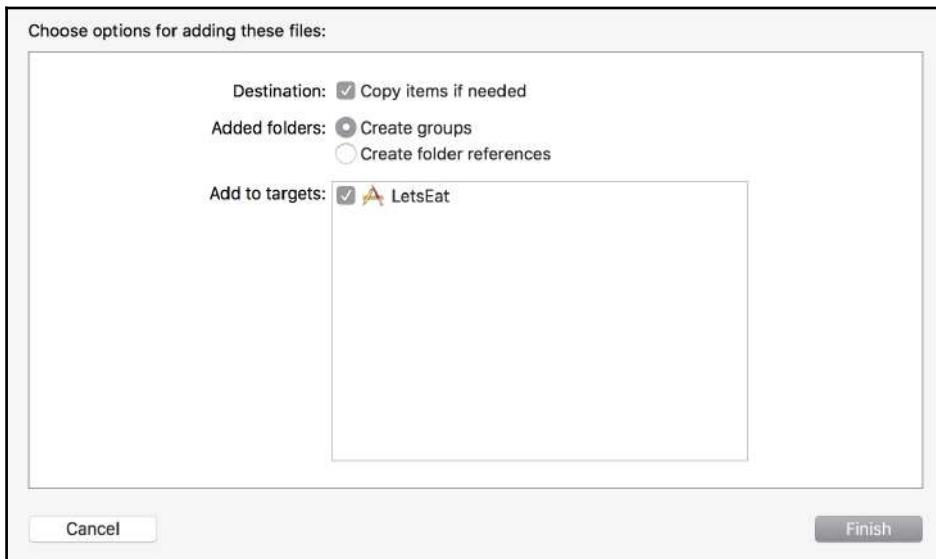


3. Select **Move to Trash**.

4. Open the project's assets folder that you downloaded from Packt's website or GitHub. Open Chapter_07. Drag the Assets.xcassets folder into your project in the Navigator Panel:

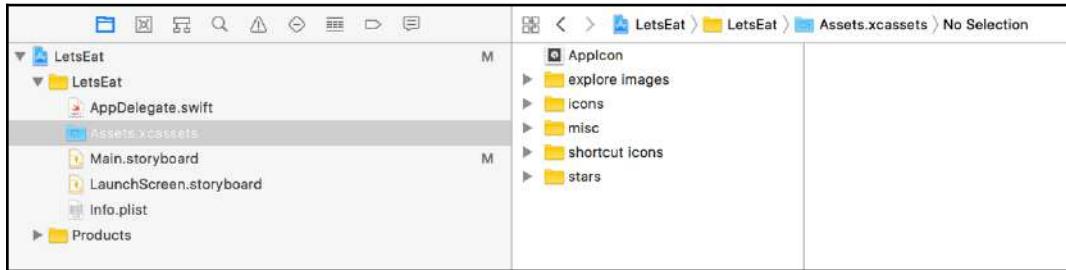


5. When you drop the folder, you will get the following message:



6. Make sure that both **Copy items if needed** and **Create groups** are selected. Then, hit **Finish**.

If you open the `Assets.xcassets` folder, you will now see all the assets for your entire project:

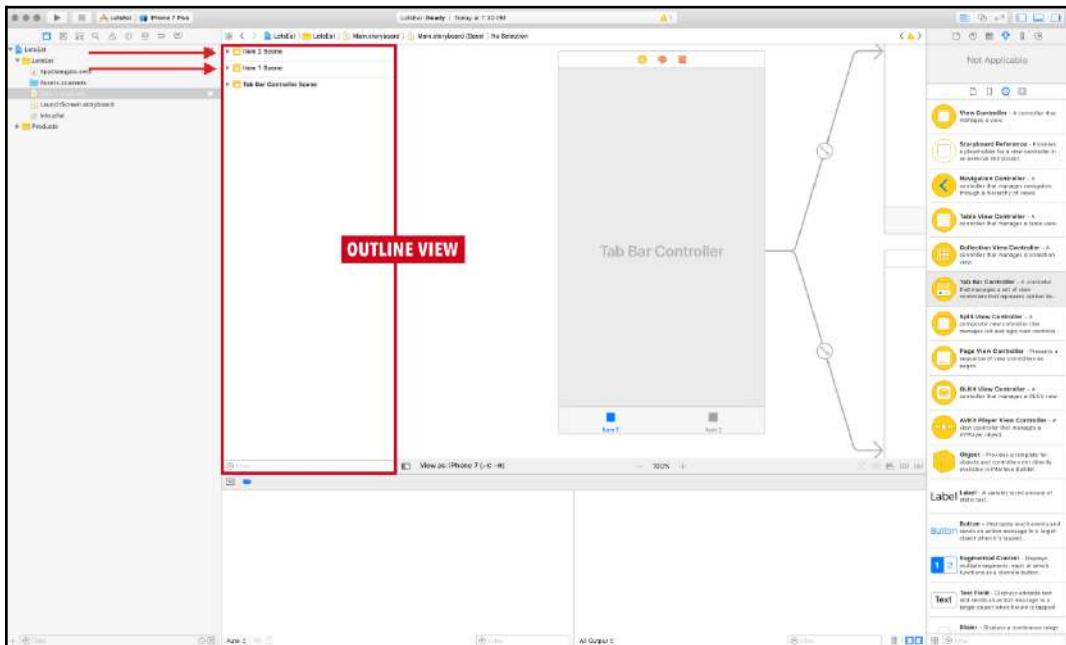


When you explore the assets, you will notice that we will be using both PNGs and PDFs.

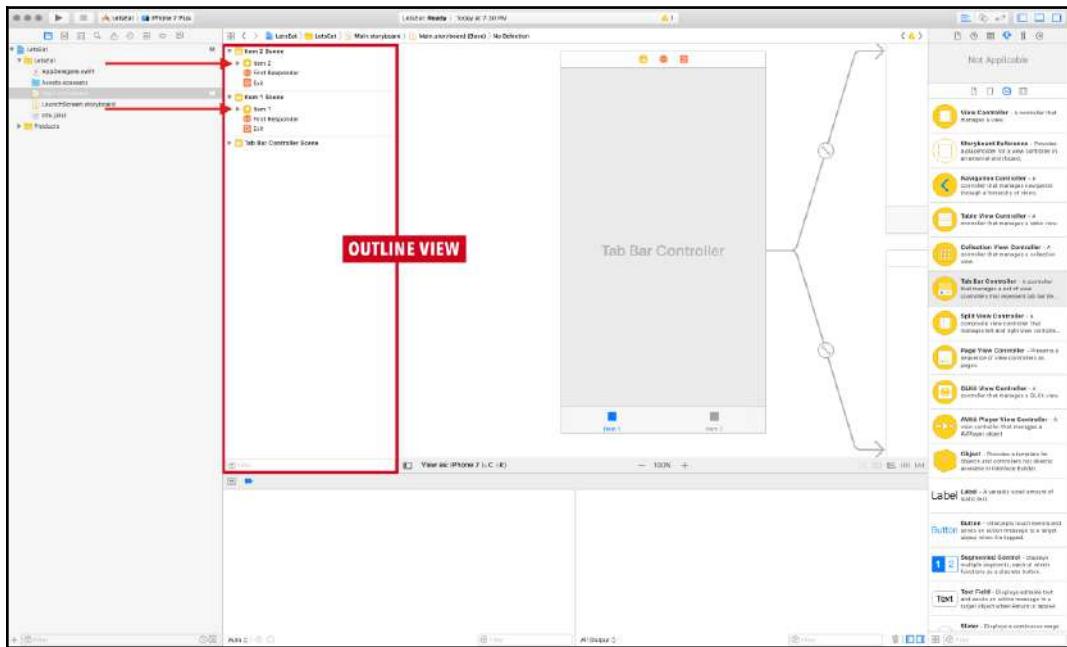
Using PDFs allows us to support multiple device resolutions with only one image. Therefore, Xcode can handle supplying assets for all resolutions.



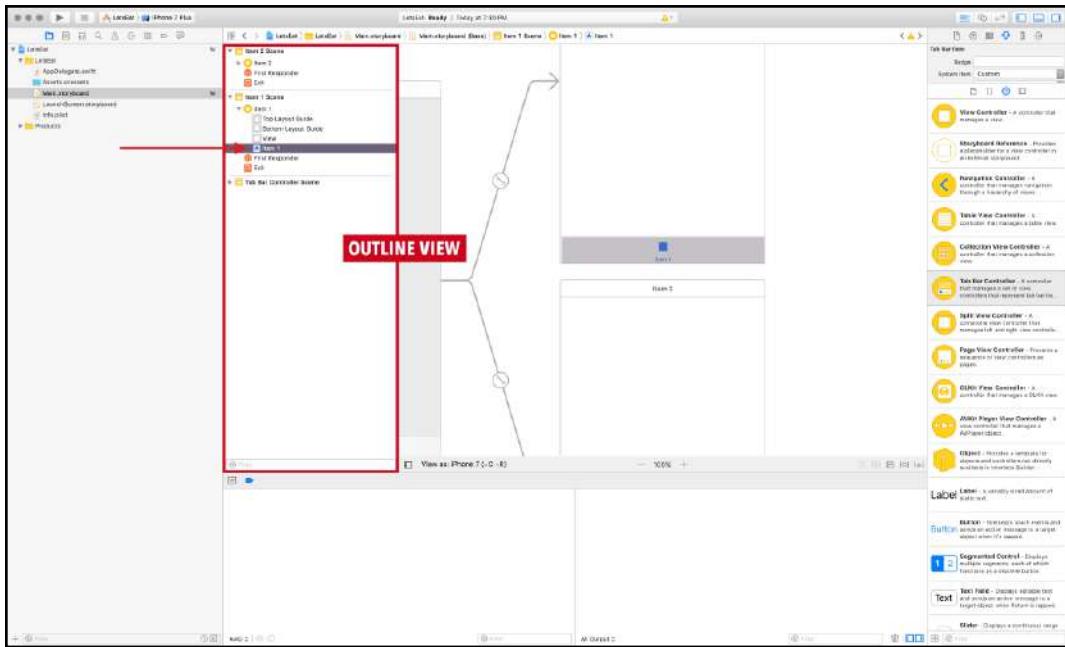
7. Select `Main.storyboard` again, and, in the Outline view, select both disclosure arrows for Item 1 Scene and Item 2 Scene, to have them face downwards:



8. Select both disclosure arrows for **Item 1** under **Item 1 Scene** and **Item 2** under **Item 2 Scene**. Both should be downward-facing:



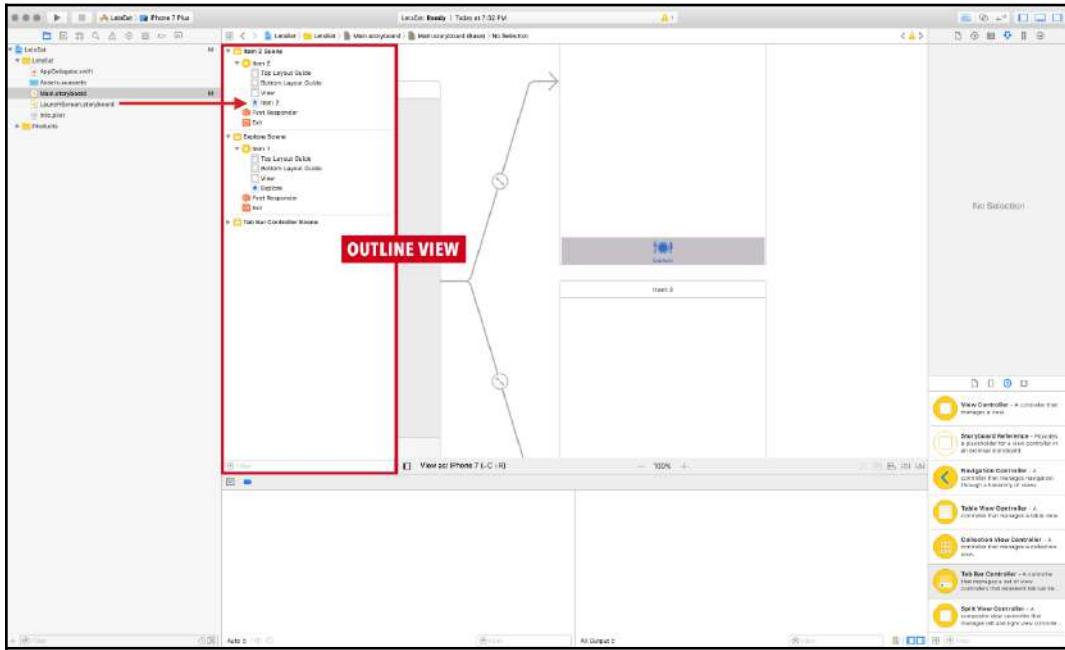
9. Select **Item 1** with the blue star to the left of it, and then select the Attributes inspector in the UTILITIES PANEL:



10. In the panel, use the following values to update your first tab icons:

- In the **Tab Bar Item**, enter the following details:
 - **Badge**: Leave this field blank
 - **System Item**: Select **Custom** from the drop-down list
 - **Selected Image**: icon-explore-on
 - **Title Position**: Select **Default Position** from the drop-down list
- In the **Bar Item**, enter the following details:
 - **Title**: Type **Explore** in this field
 - **Image**: icon-explore-off
 - **Tag**: Enter **0** in this field
 - **Enabled**: This checkbox should be checked

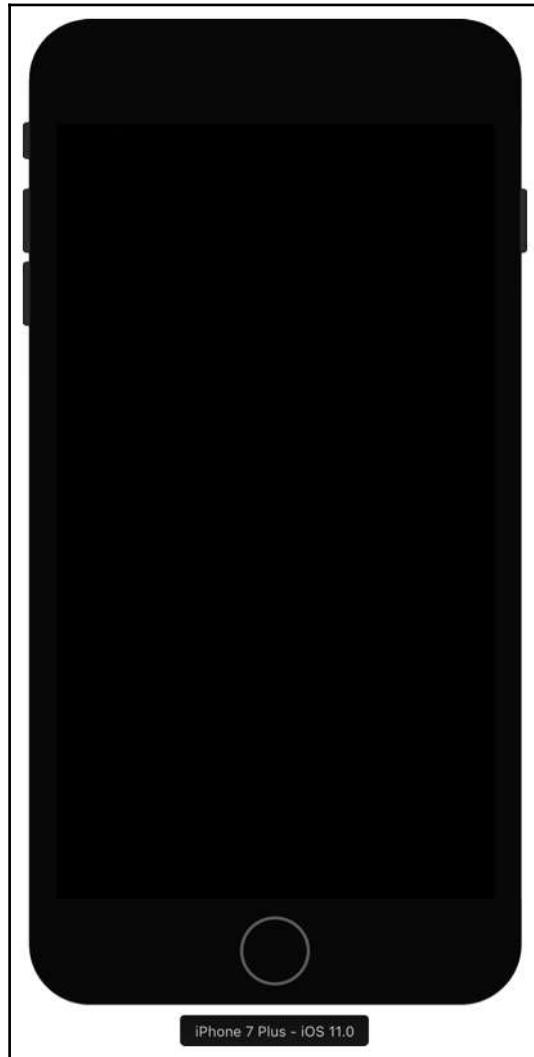
11. Select **Item 2** with the blue star to the left of it in the **OUTLINE VIEW**, and the Attributes inspector should already be open:



12. Add the following to the panel:

- In the **Tab Bar Item**, enter the following details:
 - **Badge**: Leave this field blank
 - **System Item**: Select **Custom** from the drop-down list
 - **Selected Image**: icon-map-on
 - **Title Position**: Select **Default Position** from the drop-down list
- In the **Bar Item**, enter the following details:
 - **Title**: Type Map in this field
 - **Image**: icon-map-off
 - **Tag**: Enter 0 in this field
 - **Enabled**: This checkbox should be checked

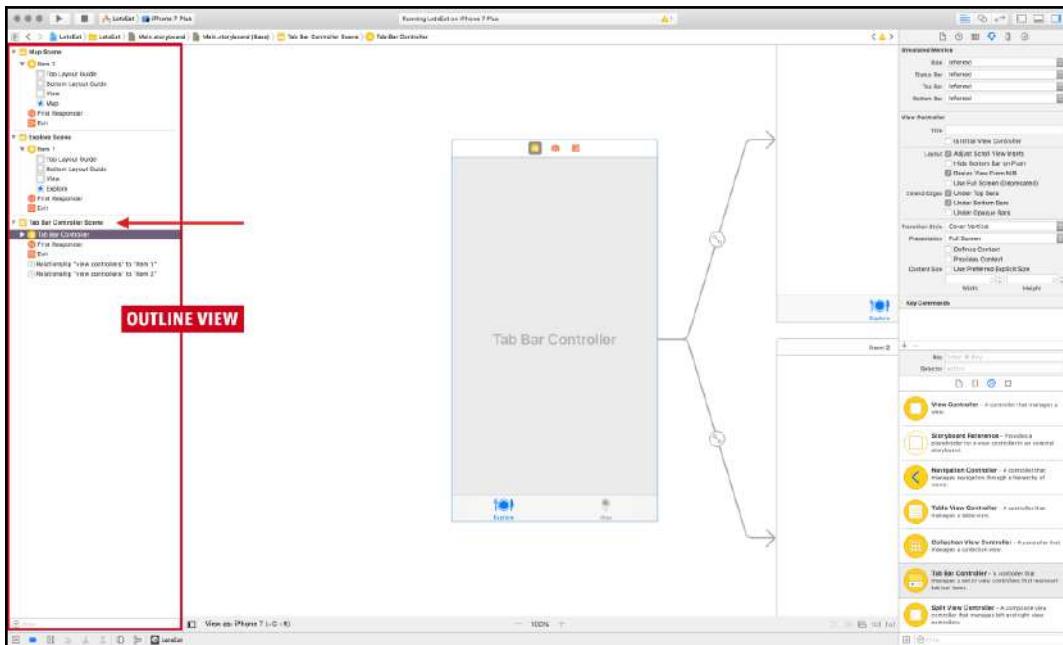
13. Run the project by hitting the play button (or use *command + R*) to see where we are:



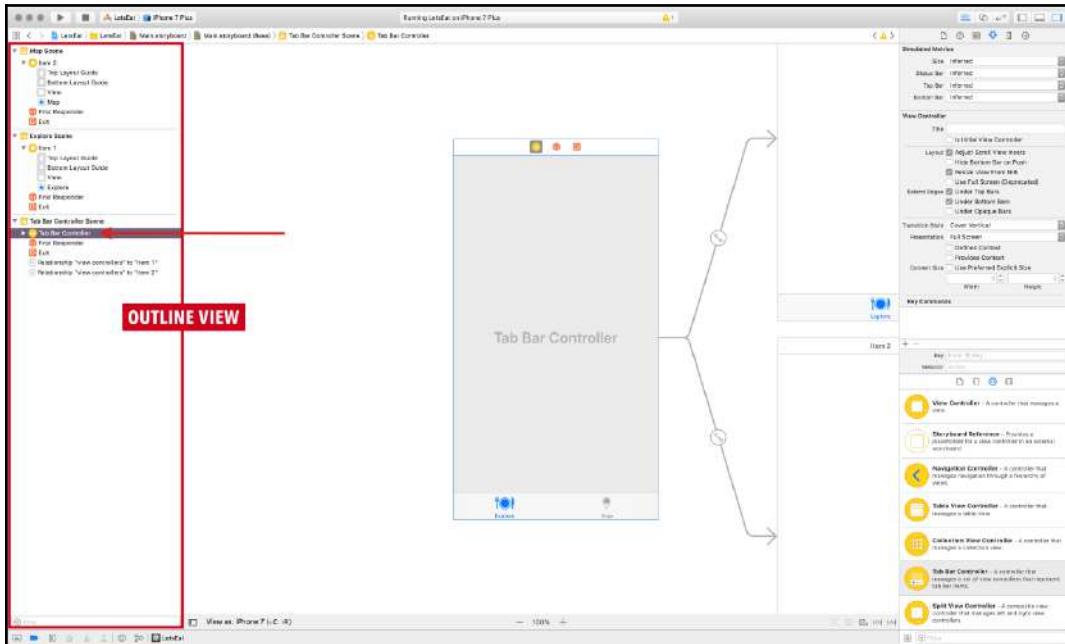
iPhone 7 Plus - iOS 11.0

As you may have noticed, this screen does not look like an app. Since we are building a Tab Bar Controller from scratch, we need to add an entry point. So, close the simulator, and continue with the steps.

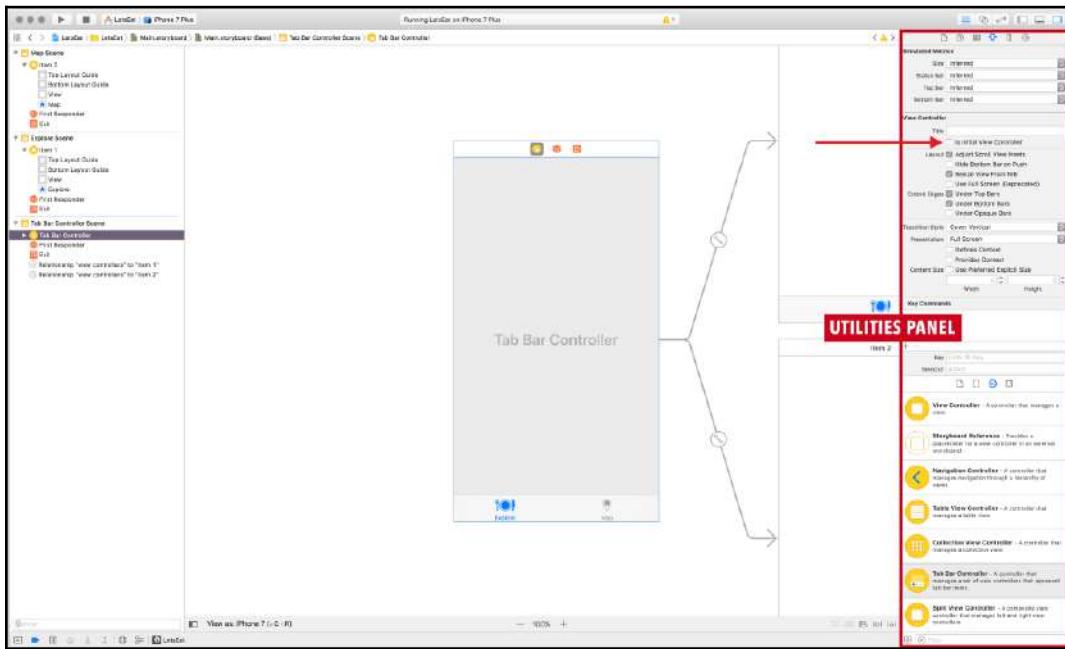
14. Select `Main.storyboard` again in the **OUTLINE VIEW**, and make sure that the disclosure arrow is down for Tab Bar Controller Scene:



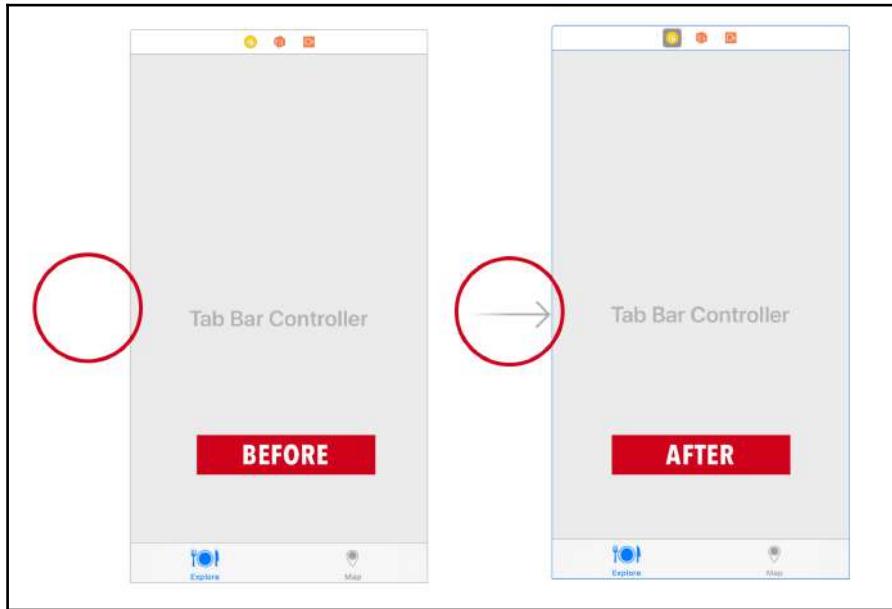
15. Select Tab Bar Controller under Tab Bar Controller Scene, and, in the **UTILITIES PANEL**, make sure that the Attributes inspector is selected:



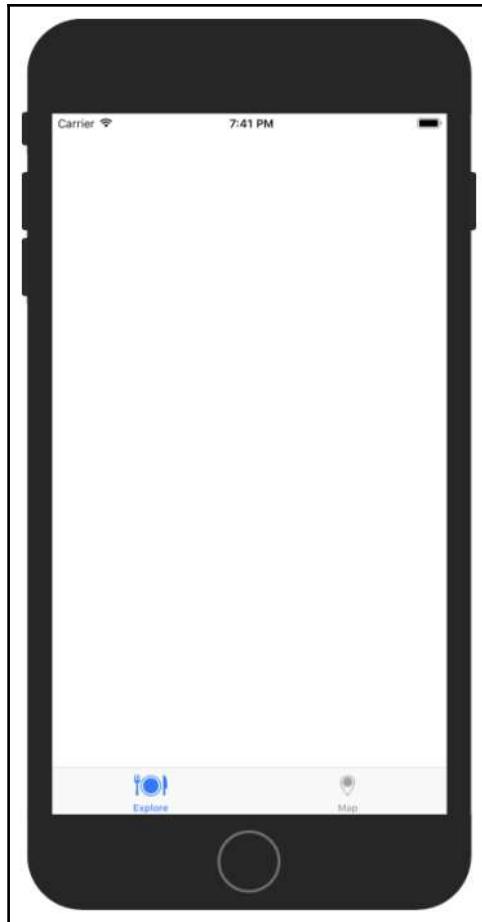
16. Under the **View Controller** section, you will need to check the box for **Is Initial View Controller**:



17. Once you set the initial **View Controller**, there will be an arrow pointing to the **Tab Bar Controller**. This arrow signifies the entry point of our app:



18. Let's rerun the project by hitting the play button (or use *command + R*):



Perfect! Now, with our basic structure established, we can start adding more specific elements to our views.

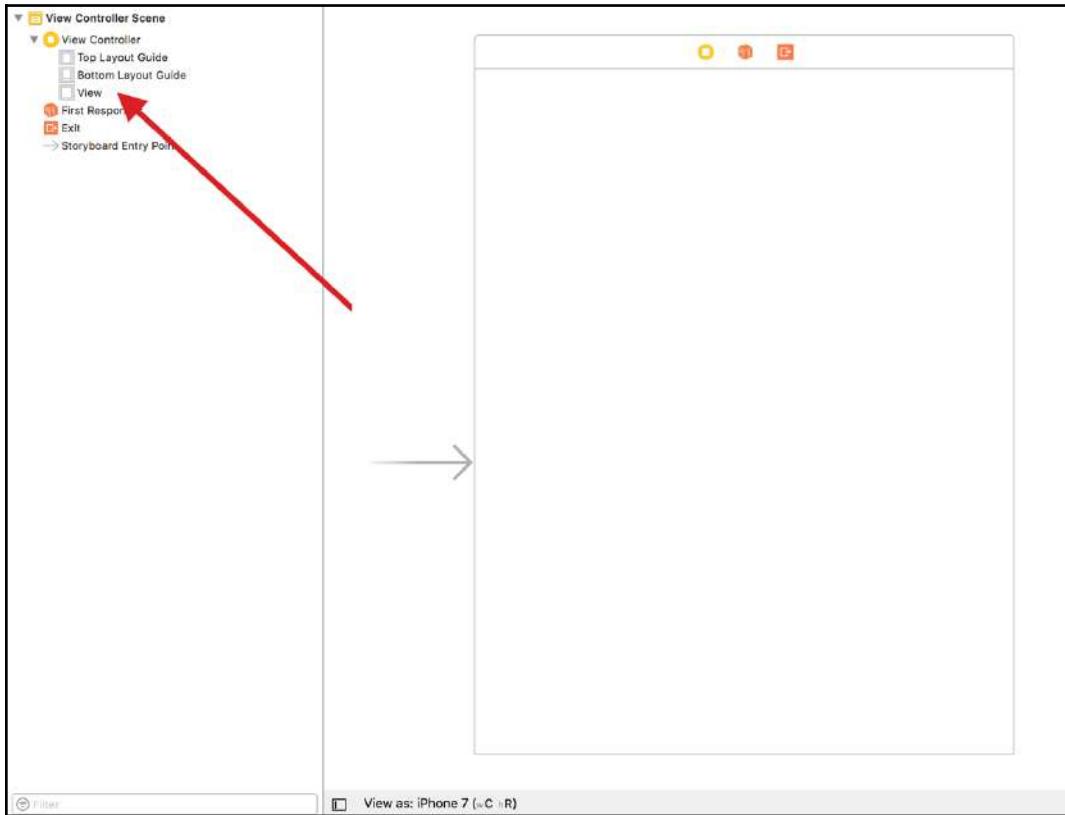
Storyboards

Before we do that, let's update `LaunchScreen.storyboard`. This storyboard is used when our app first launches.

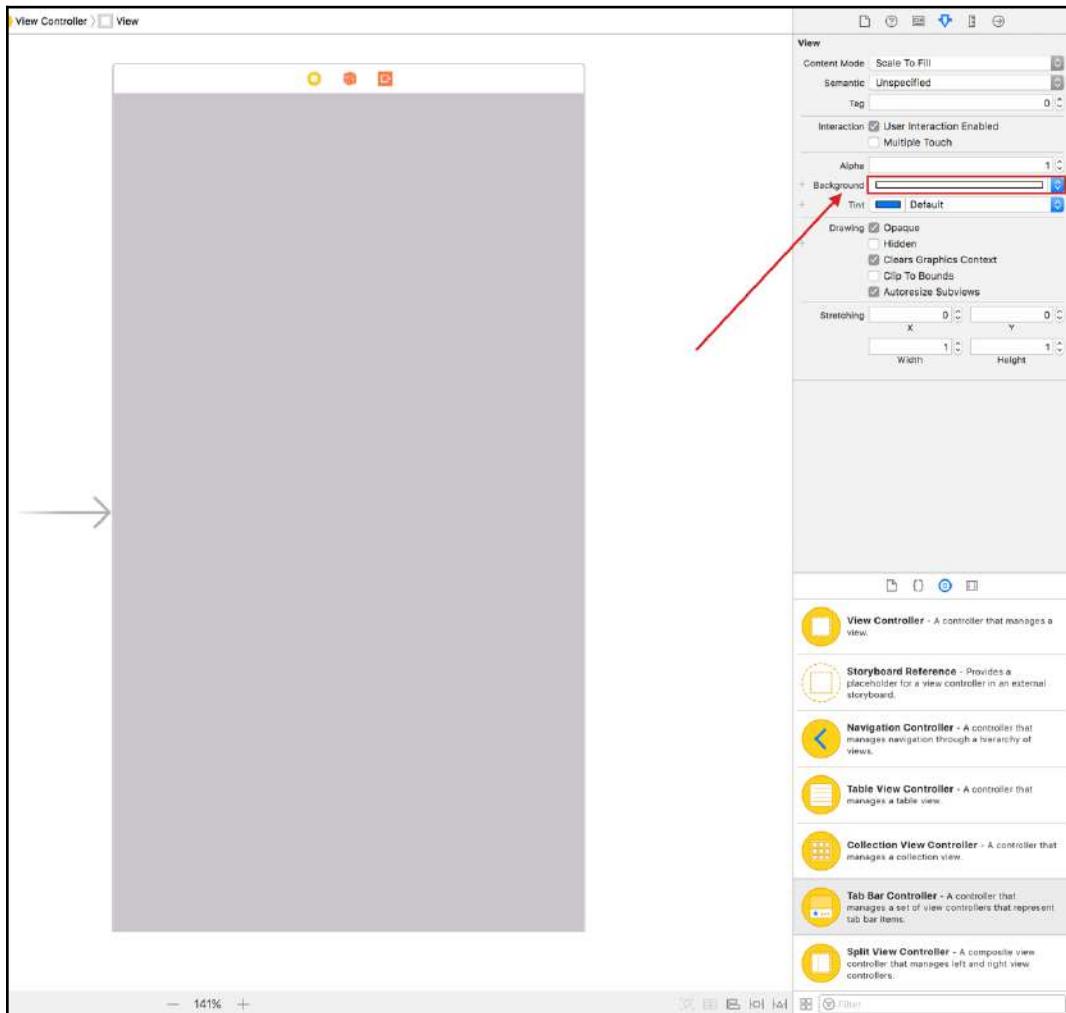
Creating our launch screen

Launch screens can use images, but that would mean that you would have to create images for every device and device orientation. Using `LaunchScreen.storyboard` gives us the ability to create just one asset for all devices and orientations:

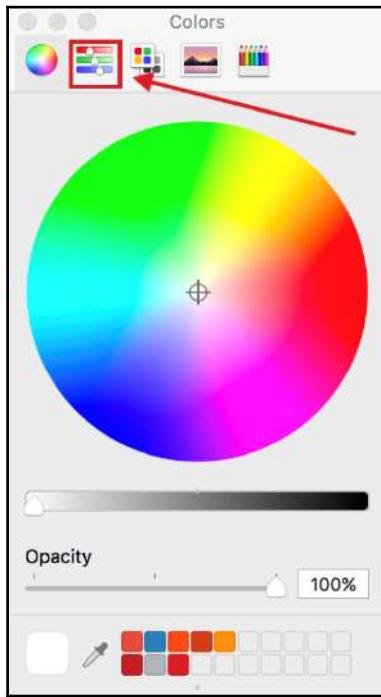
1. Select the `LaunchScreen.storyboard` file, and, in the Outline view, make sure that the disclosure arrows for View Controller Scene and View Controller are collapsed. Then, select View under View Controller:



2. In the UTILITIES PANEL panel, select the Attributes inspector, and click on the white **Background** bar:



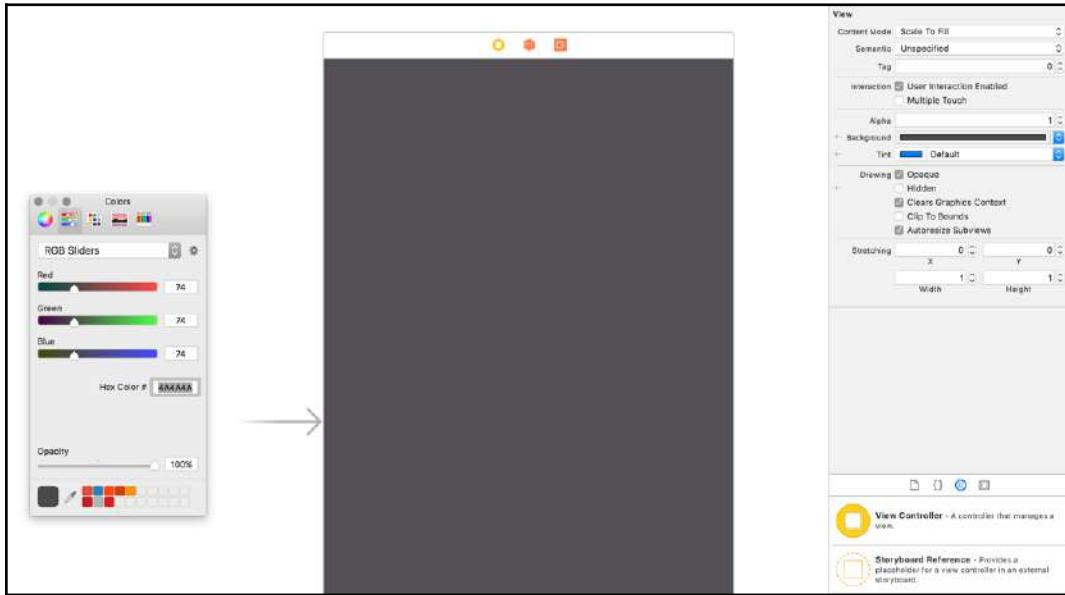
3. A **Colors** panel will appear. Select the second tab, which is called the **Color Slider**:



- Under **RGB Sliders**, **Hex Color #**, update the value from FFFFFF to 4A4A4A. This should change your background color from white to a dark gray:

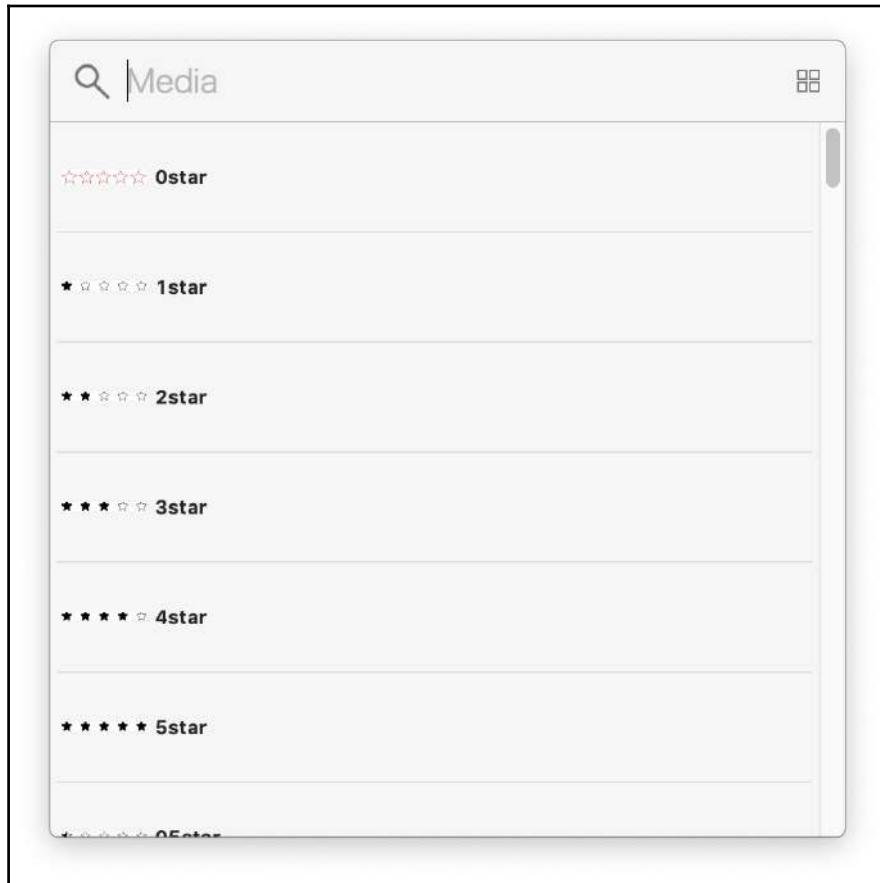


5. You might have to select the background color a second time. If so, select the **Background** bar in the Attributes inspector again, which should change the **Hex Color #** back to FFFFFF. Then, change it again to 4A4A4A. You can now close the **Color** panel, and you should see the background color update in your Standard Editor panel:



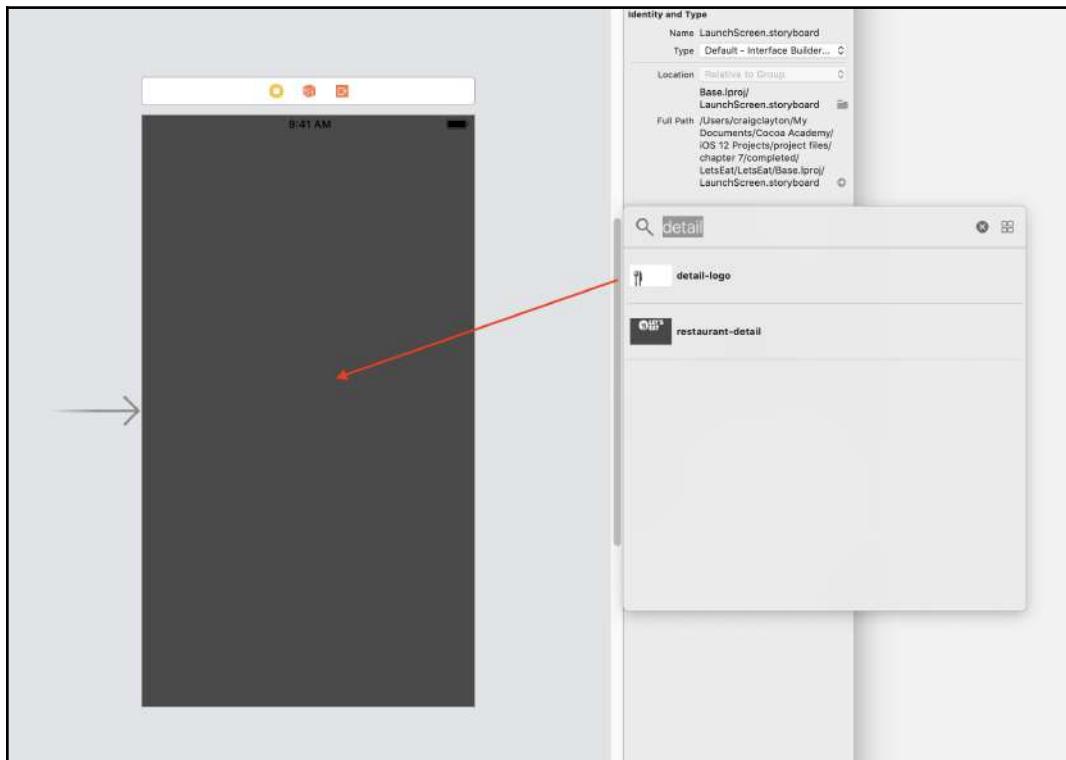
Next, we need to bring the app logo onto the screen:

1. While still in `LaunchScreen.storyboard`, launch the Media Library (`command + Shift + M`). You can also access the library if you long-press the object library, but the shortcut is much faster:



The Media Library allows us to access our image assets, and it will place them inside a `UIImageView` for us.

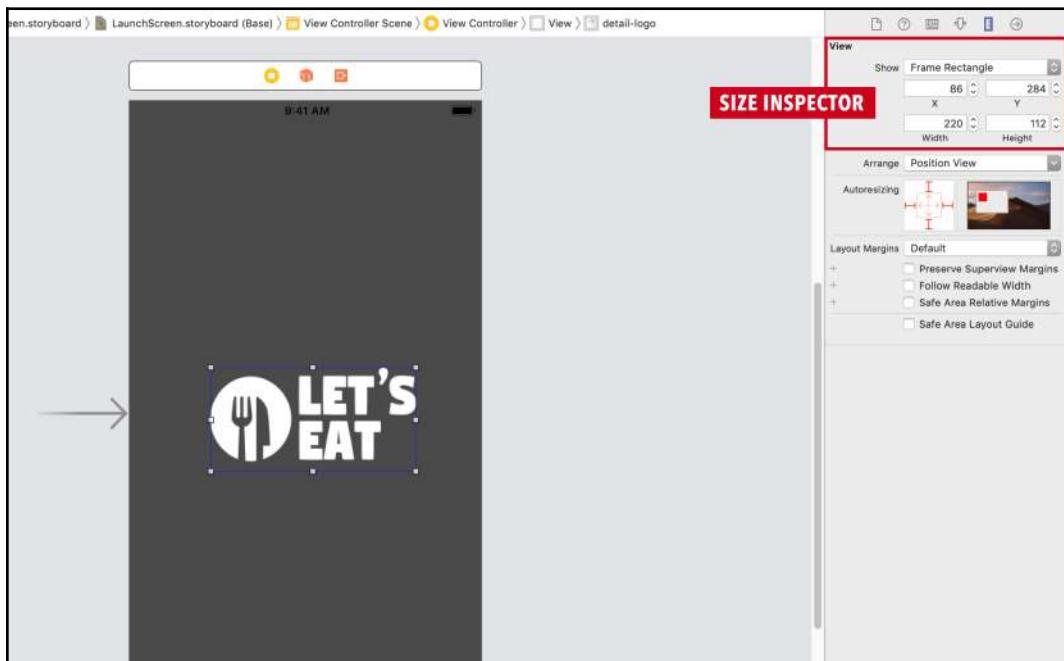
2. In the filter, at the bottom of the Library pane, type `detail-logo`. Once that appears, drag and drop the logo onto `LaunchScreen.storyboard`:



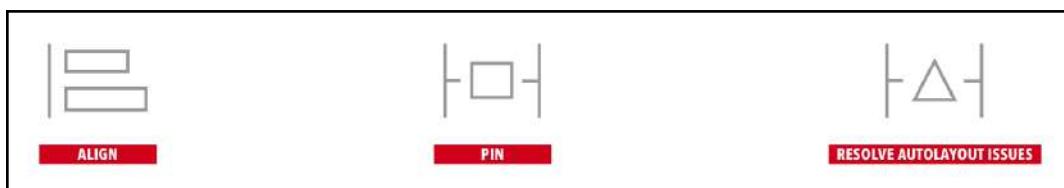
There might be a bug in Xcode and, therefore, sometimes when you drag the logo out, the width and height will not be set, and you will need to enter the width and height manually.

3. If your logo does not drag out to size, do this step: with the logo selected, open the **SIZE INSPECTOR** in the **UTILITIES PANEL**, and set the width and height to the following:

- **Width:** 220
- **Height:** 112



4. We want our *Let's Eat* logo to appear in the center of the screen. For our logo to appear in the center for all devices, we need to apply auto layout. Select `detail-logo`, and then select the **ALIGN** icon, which is to the left of the **PIN** icon:



5. Check the following boxes that appear:

- Horizontally in container
- Vertically in container

6. Click on **Add 2 Constraints**.

When you are done, you will see the following:



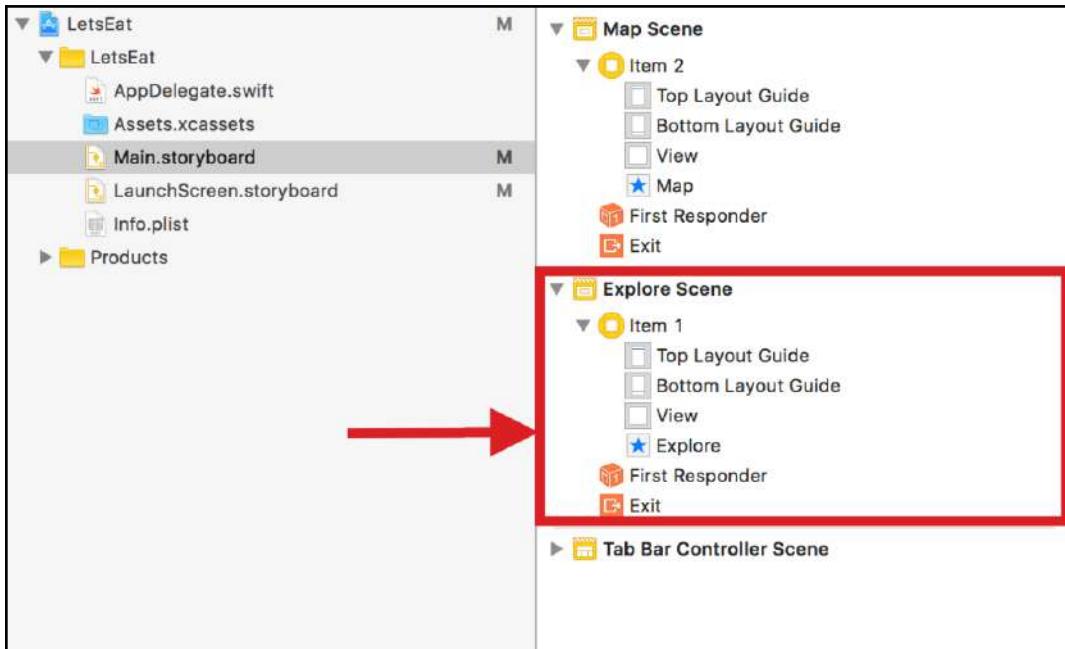
Our launch screen is now set up for all devices. If you rerun the project, you will now see the launch screen with the **LET'S EAT** logo and new background color.

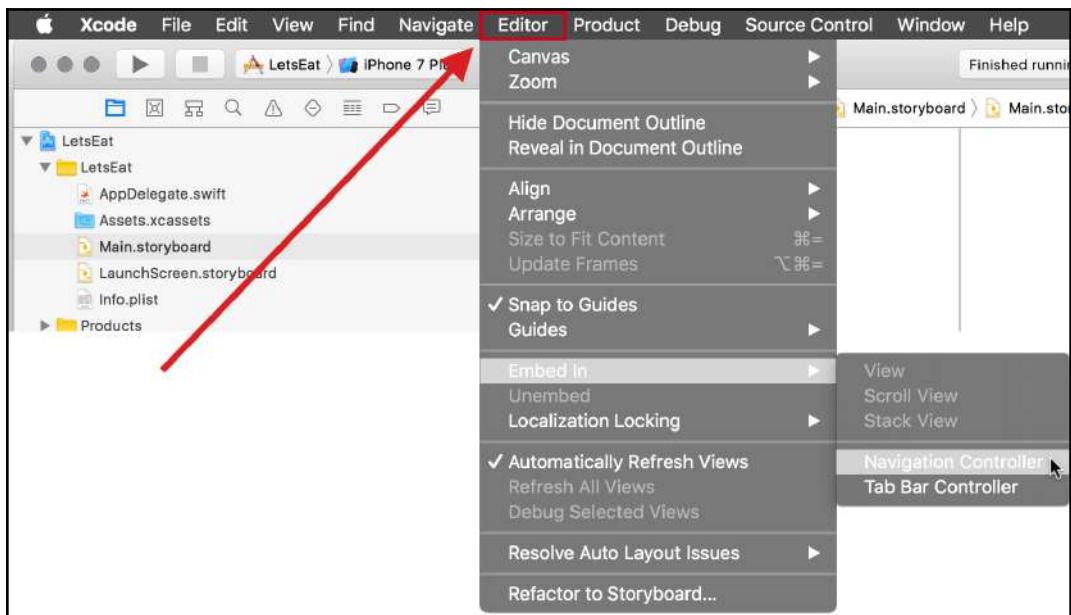
Let's move on to adding some detail to our **Explore** tab, since this is the first thing a user will see after the app launches.

Adding a Navigation Controller

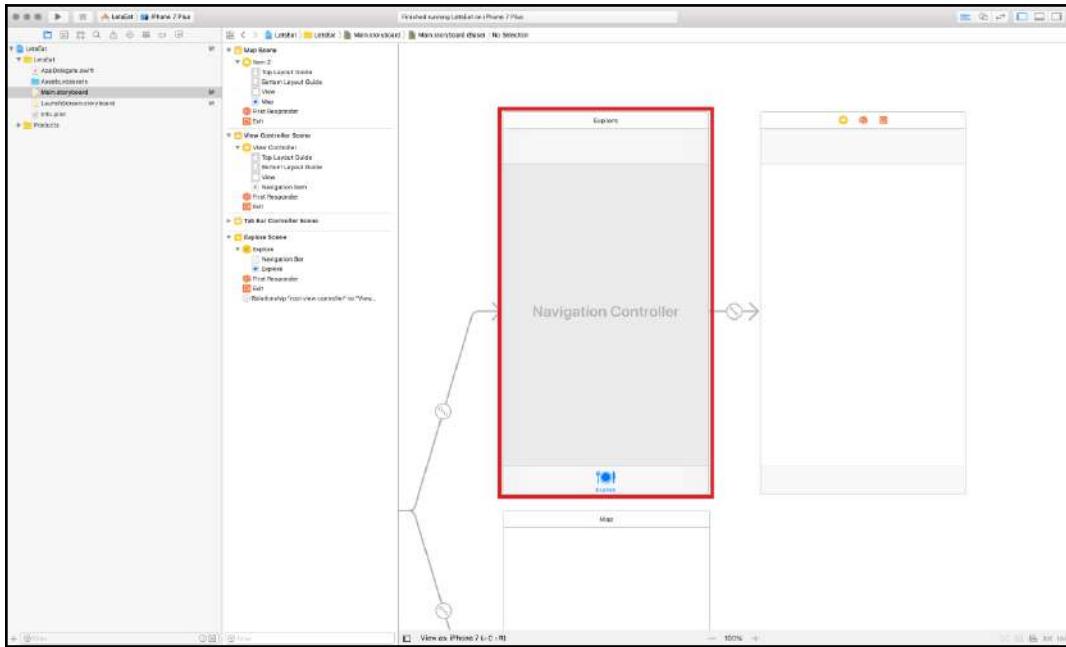
We first need to add a Navigation Controller to our **Explore** tab. The Navigation Controller will allow us to do a few things, such as adding a button to the title bar of the navigation to present our cities list:

1. Select `Main.storyboard`, and, in the **OUTLINE VIEW**, select `Explore` with the blue star to the left of it, under `Item 1` in `Explore Scene`:

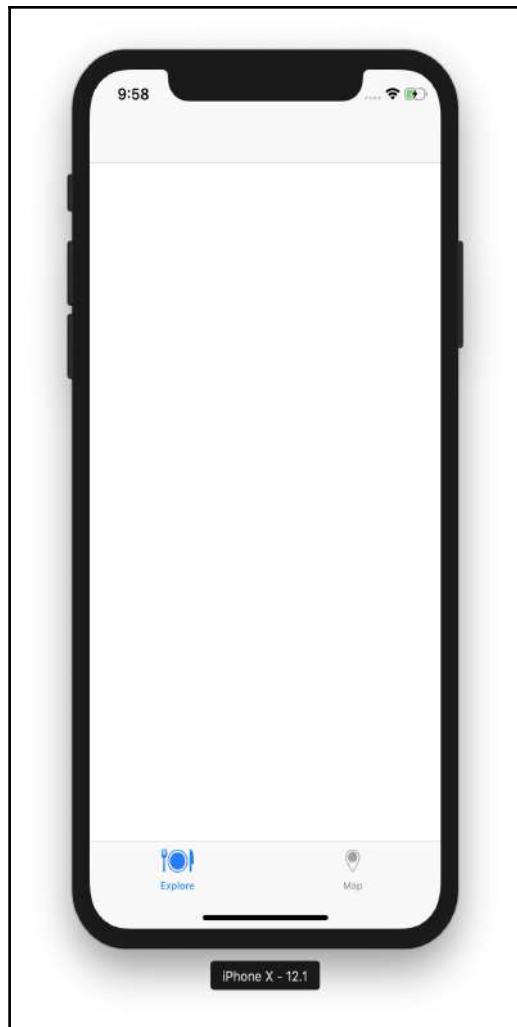


2. Navigate to Editor | Embed In | Navigation Controller:

3. Our View Controller has a Navigation Controller:



4. Run the project by hitting the Play button (or use *command + R*):



Repeat steps 1 to 4 from the *Adding a Navigation Controller* subsection for the **Map** tab. Now that we have added both Navigation Controllers, in the next chapter, we will continue to create other View Controllers.

Summary

Storyboarding is one of the things I enjoy doing. It is quick and easy to set up your UI with storyboards. Being able to drag and drop what you need onto the canvas is such an efficient method of developing app storyboards. There are times when you will need to code, but being able to work on things without having to write any code is an excellent capability. My preference is to use storyboards as much as possible, but many developers prefer to work in code. If you come from another programming language, try to keep an open mind and learn storyboarding.

When you work on a project that uses storyboards, you can get a high-level overview of the project. When everything is written in code, it takes more time to get a basic idea of how the app is structured, and its overall flow. Again, some people love to code their UI, and we will do some of that in this book. My main point is that you have to find what works for you. This book leans more toward the storyboard side versus the coding side of setting up your UI.

In the next chapter, we will continue setting up our UI, and become familiar with more of the UI elements that you have seen in many iOS apps.

8

Building Our App Structure in Storyboard

In the previous chapter, we created our Tab Bar Controller. In this chapter, we will be creating other View Controllers that we need in our app. Our goal for the end of this chapter is to be able to navigate through the app with the least code required.

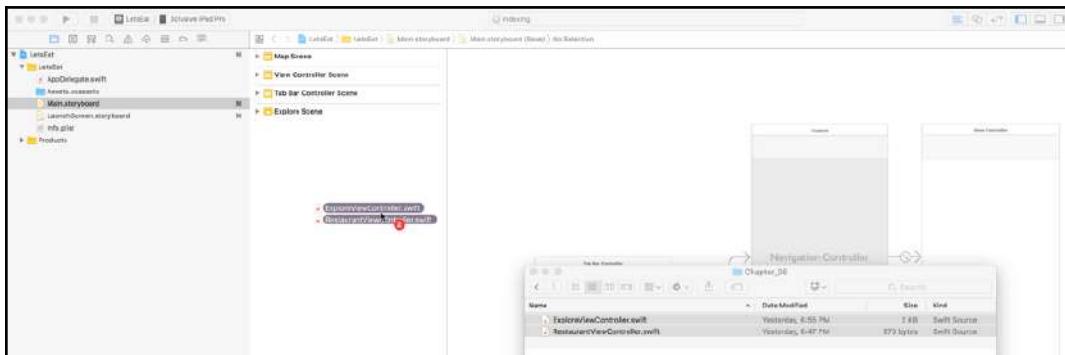
The following will be covered in this chapter:

- Collection View
- Outlets
- Modals

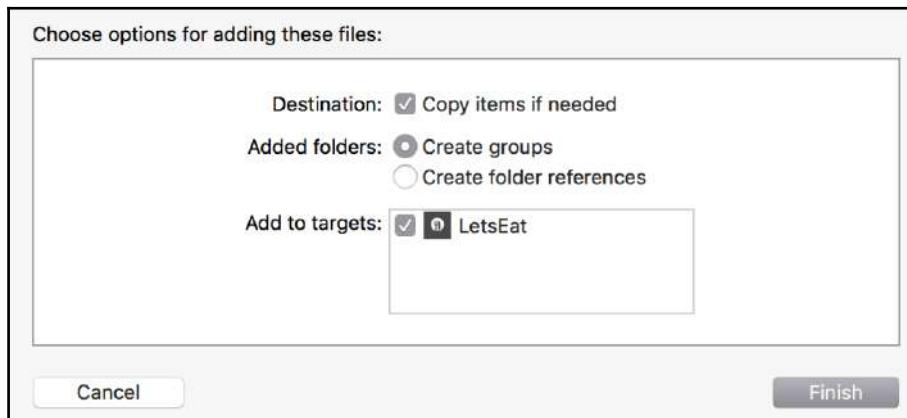
Before we begin setting up our Collection View Controller, you will need to add two files, `ExploreViewController` and `RestaurantViewController`, which you'd have downloaded from Packt's website or GitHub. By combining these files and then a bit of code, we will be able to focus on the design of our app.

Later in the book, we will delete these files, and create them ourselves. But, for this chapter, let's add these two files into our project:

1. Open the assets project folder that you downloaded from Packt's website or GitHub. Open Chapter_08 and drag the two files in the folder into your project in the Navigator panel:



2. When you drop the folder, you will get the following message:



3. Make sure you have both **Copy items if needed** and **Create groups** selected, then hit **Finish**.
4. Add code to these new files, which will allow us to dismiss modals that we will create later in this chapter. A modal is a container that opens on top of the current content showing in an app and allows you to take more action without opening up all of the information on the viewed screen.
Let's add the code to enable us to dismiss modals. Open the `ExploreViewController.swift` file and, under where it says `// Add Unwind here` at the bottom of the file, add the following code:

```
@IBAction func unwindLocationCancel(segue:UIStoryboardSegue) { }
```

```
// ViewController.swift
// CollectionViewTest
//
// Created by Craig Clayton on 6/30/17.
// Copyright © 2017 Cocoa Academy. All rights reserved.

import UIKit

class ExploreViewController: UIViewController {

    @IBOutlet weak var collectionView: UICollectionView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let layout = UICollectionViewFlowLayout()
        layout.headerReferenceSize = CGSize(width: 0, height: 100)
        layout.sectionHeadersPinToVisibleBounds = true
        collectionView.collectionViewLayout = layout
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    func collectionView(_ collectionView: UICollectionView, viewForSupplementaryElementOfKind kind: String, at indexPath: IndexPath) -> UICollectionReusableView {
        let headerView = collectionView.dequeueReusableCellSupplementaryView(ofKind: kind, withReuseIdentifier: "header", for: indexPath)
        return headerView
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        return collectionView.dequeueReusableCell(withReuseIdentifier: "exploreCell", for: indexPath)
    }

    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return 1
    }

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return 20
    }
}

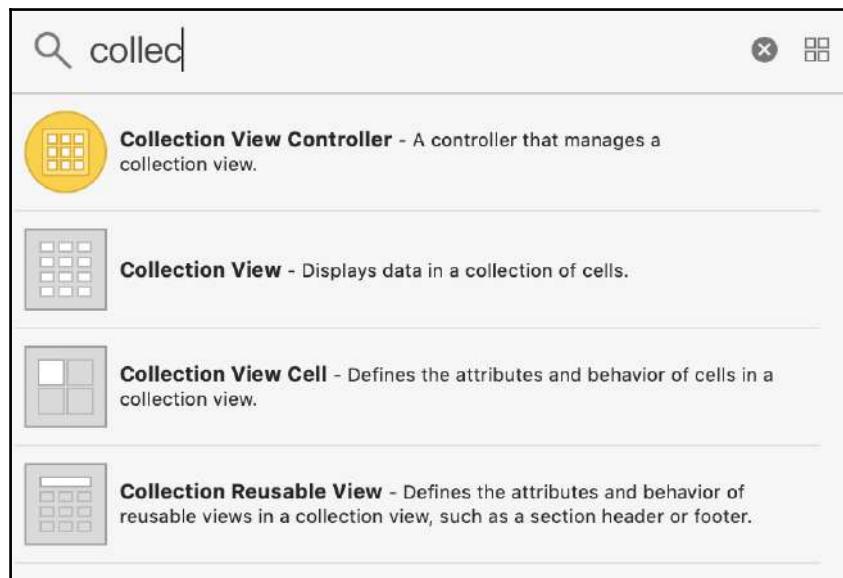
// Add Unwind here
```

If we look at our app design, which we reviewed earlier in this book, in our first tab, the **Explore** tab, we show a grid of food cuisines as well as a list of locations. First, we will set up our grid.

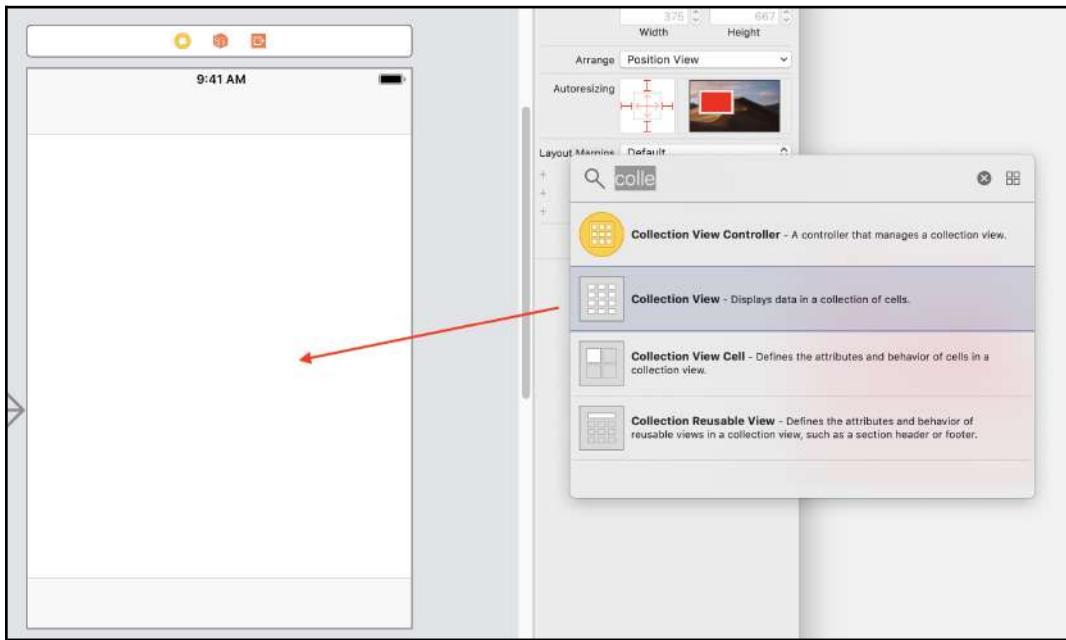
Adding a Collection View Controller

As we discussed earlier in the book, Collection View Controllers allow us to display elements within a grid. Let's set up our **Collection View**:

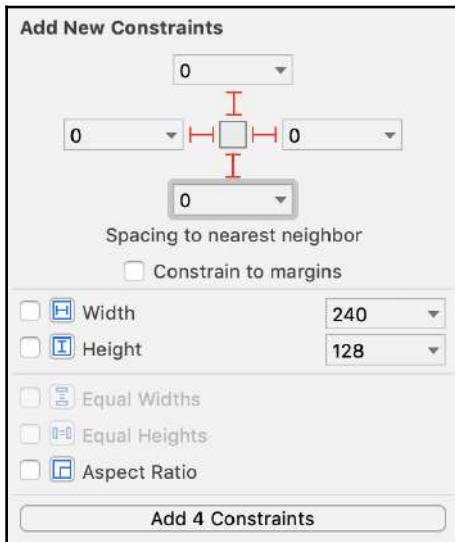
1. Select the `Main.storyboard` file, making sure that you are zoomed out and can see all of your scenes. In the Utilities Panel (*command + Shift + L*), ensure that you have the object library tab selected.
2. In the filter field, type `collec`:



3. Click on and drag **Collection View**, and drop it onto the **Explore View Controller**:



4. You will see small boxes around the entire **Collection View** component. Select the Pin icon, and enter the following values:



All values under **Add New Constraints** are set to 0.

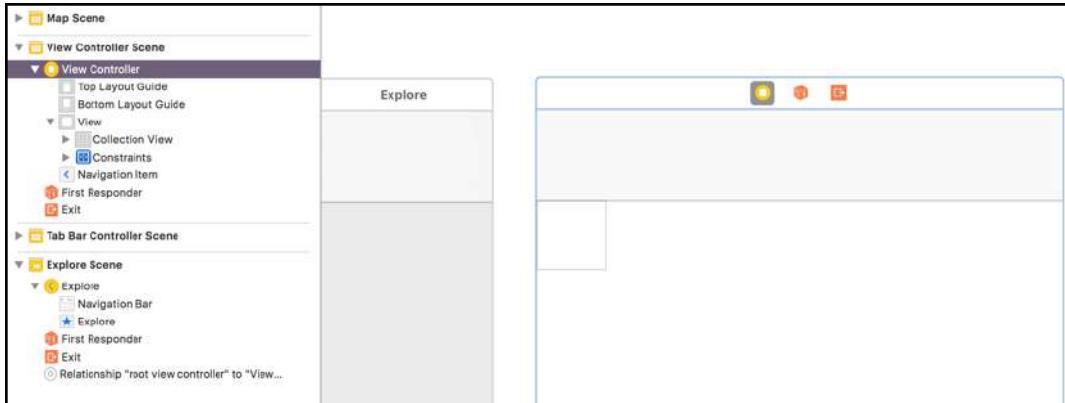
5. Click on **Add 4 Constraints**.

We now have our **Collection View** component set up for our **Explore** tab.

Hooking up our outlets

Let's now link our file, `ExploreViewController`, to our `UIViewController` in the storyboard:

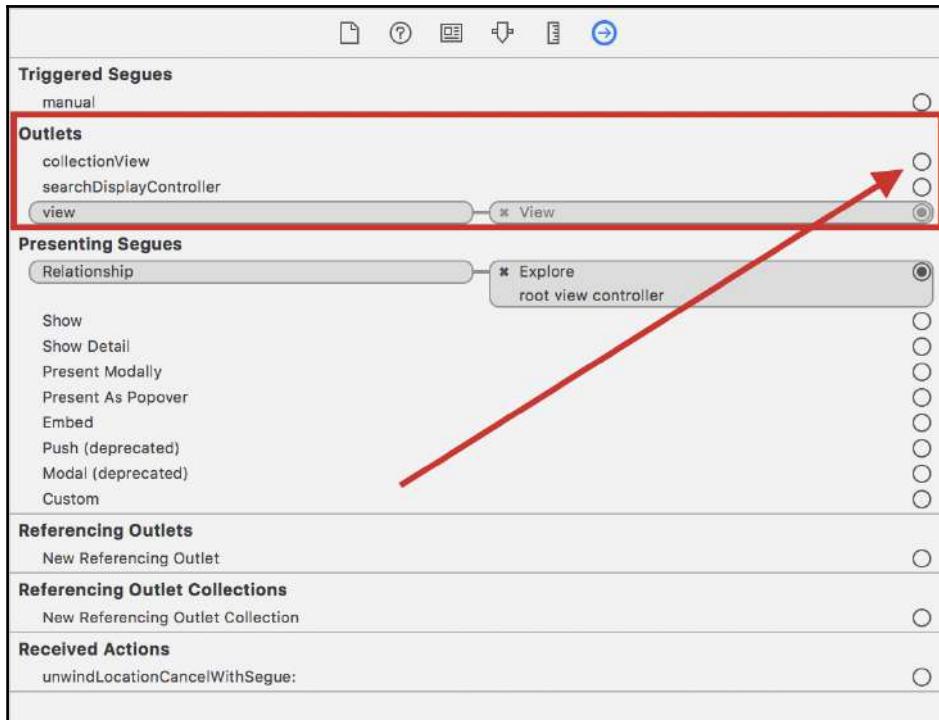
1. While still in the `Main.storyboard` file, select the `UIViewController` with the **Collection View** that we just created, by clicking on the leftmost icon at the top of that controller:



2. In the Utilities Panel, select the Identity Inspector, which is the third icon from the left:

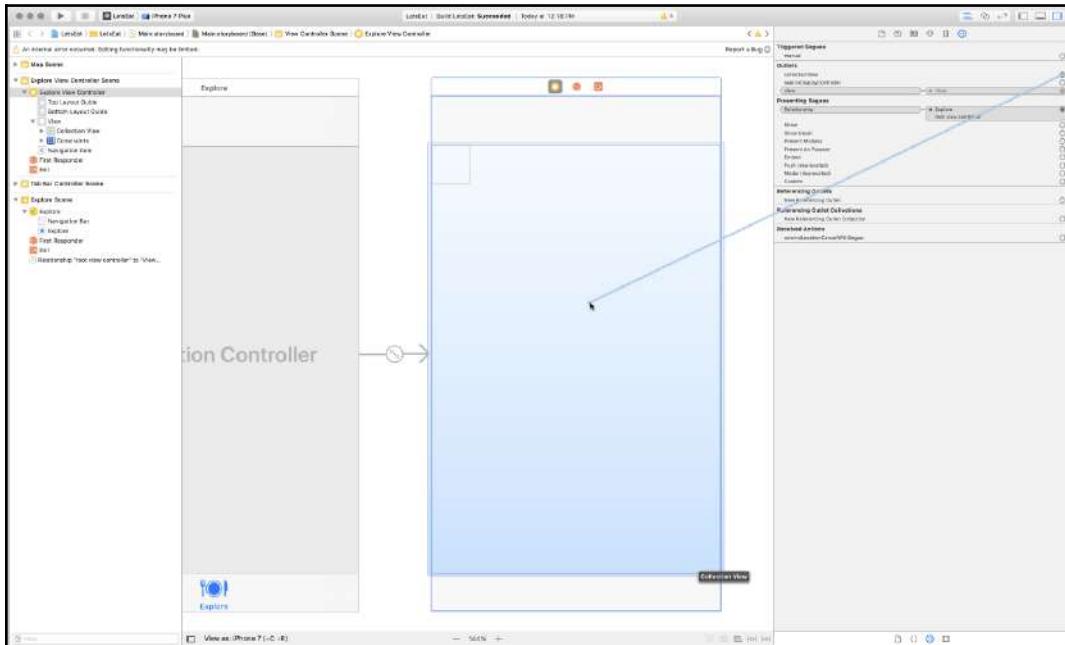


3. Under **Custom Class**, in the **Class** drop-down menu, choose `ExploreViewController` and hit the *Enter* key.
4. Select the **Connections Inspector**, the last icon on the right, in the Utilities Panel.
5. Under **Outlets**, you will see **collectionView** and an empty circle:

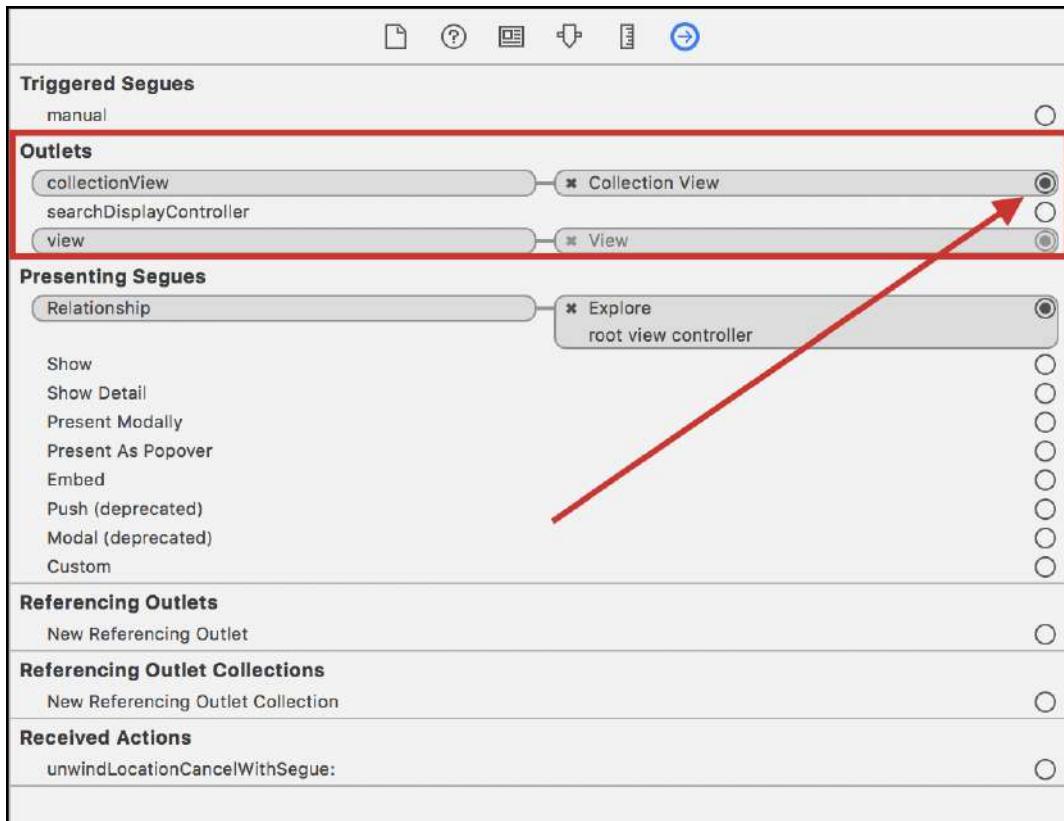


IBOutlet is a way to connect to a UI element. We have a **Collection View** on our `UIViewController`; now, we are hooking into that variable. Later in the book, you will learn how to create these variables.

6. Click on the **collectionView** circle, and click-drag from the circle in the Connections inspector to the **Collection View** that we just added inside of the **UIViewController**:



7. Release it and you will see the circle become filled:

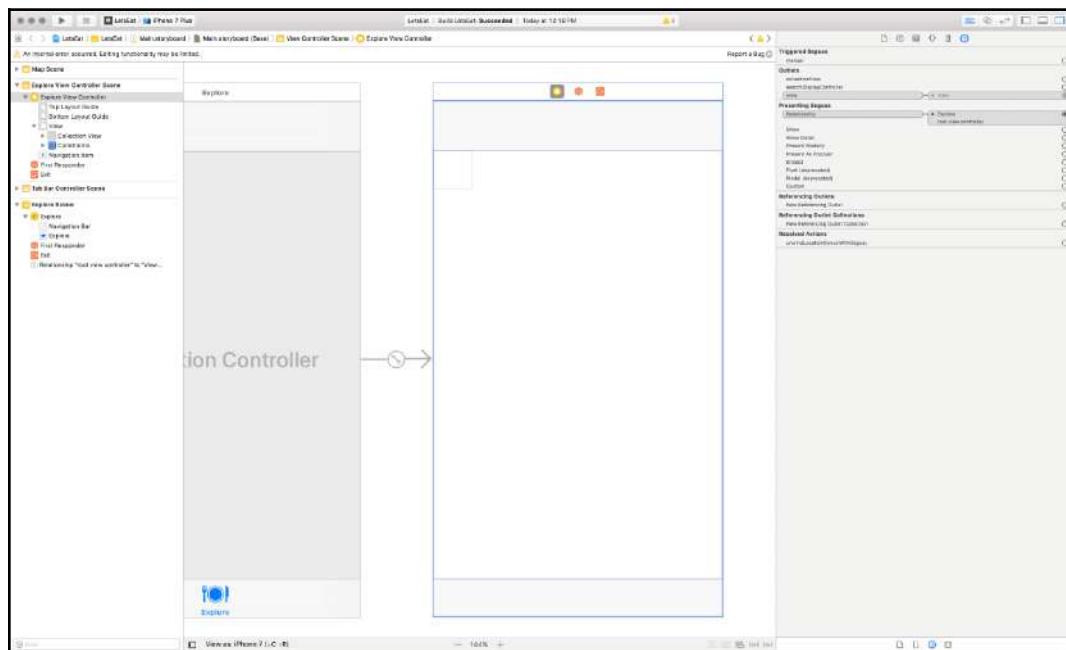


We need to hook up the data source and delegate. The delegate and data source allow us to pass data to our **Collection View**, and to know when our **Collection View** has some kind of interaction.



The `dataSource` property is what is used to supply the data for our **Collection View**, so we need to pass whatever data we have to this property. On the other hand, the `delegate` property, which supplies the behavior, does not require us to provide anything, as it receives interactions that happen within our **Collection View**.

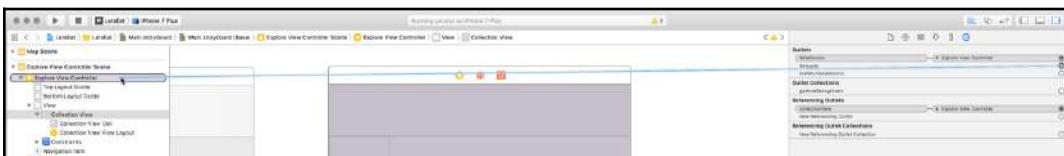
8. In your scene, select your **Collection View** and then, in the Utilities Panel, select the **Connections Inspector**.
9. Under the **Outlets** section, you will see two empty circles, `dataSource` and `delegate`:



10. Click and drag from the empty circle of the `dataSource` property to the **Explore View Controller** in your Outline view, and then release:



11. Repeat for the `delegate` property:



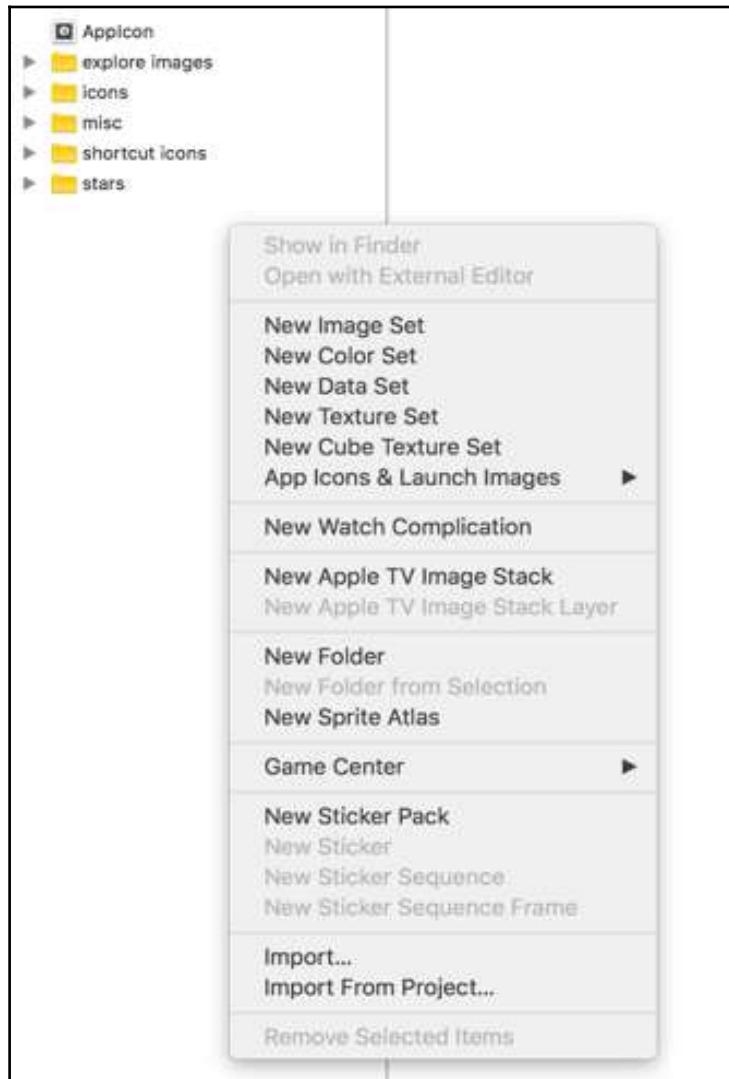
Next, let's set up our **Collection View** prototype cell to have a color.

Creating a custom color

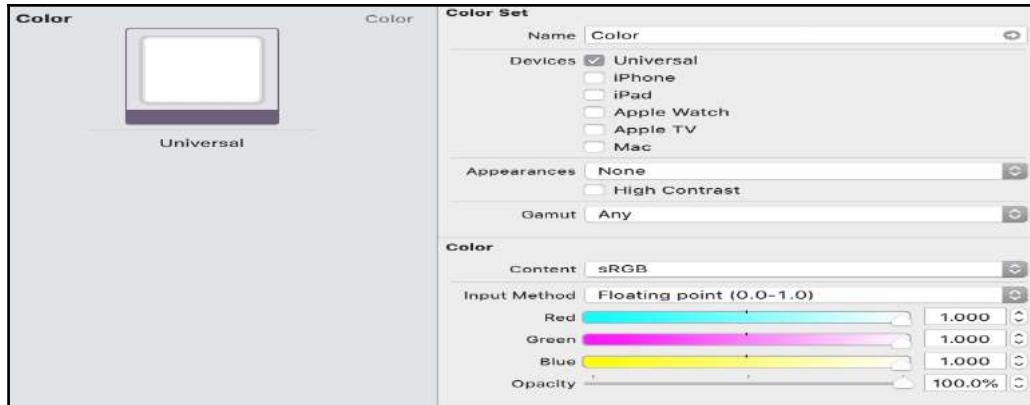
Let's add colors to your `Assets.xcassets` folder. Adding colors here is great when you want to have all your colors in one location. Before we update our explore cell, let's create a new color:

1. Open the `Assets.xcassets` file.

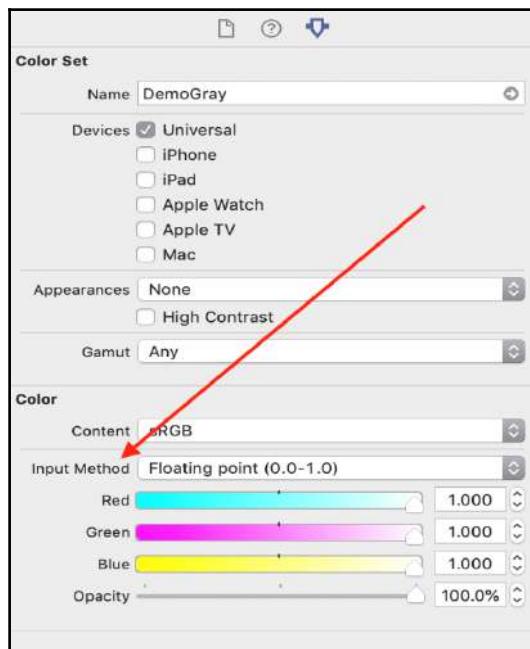
2. Right-click inside of Assets.xcassets, where you will see folders, and create a new folder called colors:



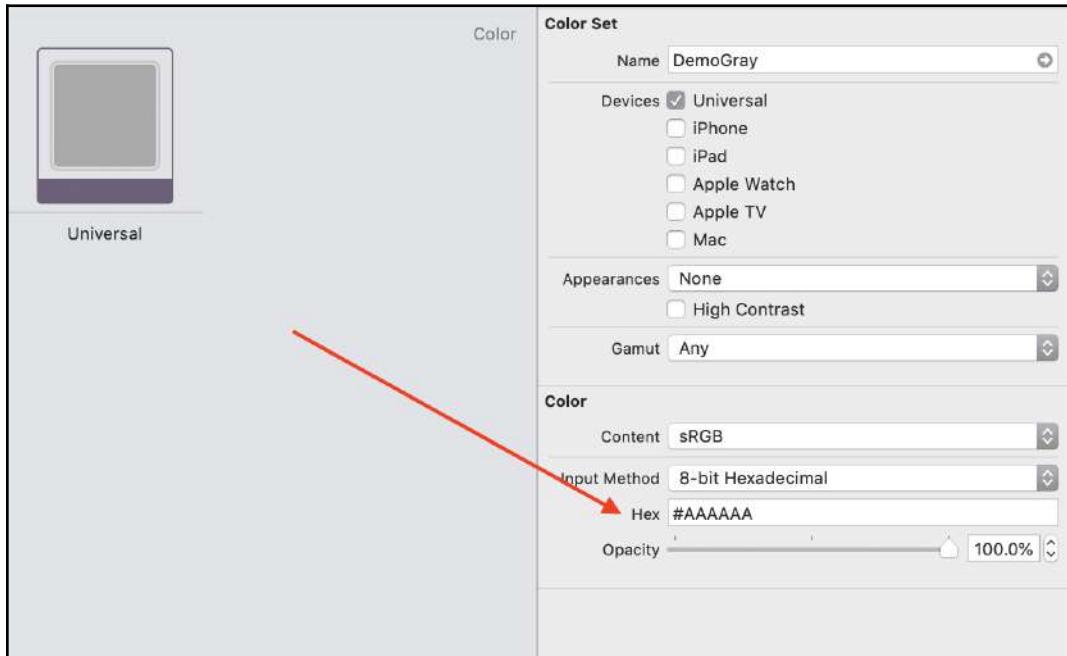
3. Right-click the `colors` folder, and, this time, select **New Color Set**. You will see a new color added to your folder. Select the Attributes inspector in the Utilities panel:



4. Under **Color** set, update the name to Demo_Grey.
5. Under **Color**, click the **Input Method** dropdown:



6. Select 8-bit Hexadecimal. Change the **Hex #** value to **#AAAAAA**. When you are done, you should see the following:

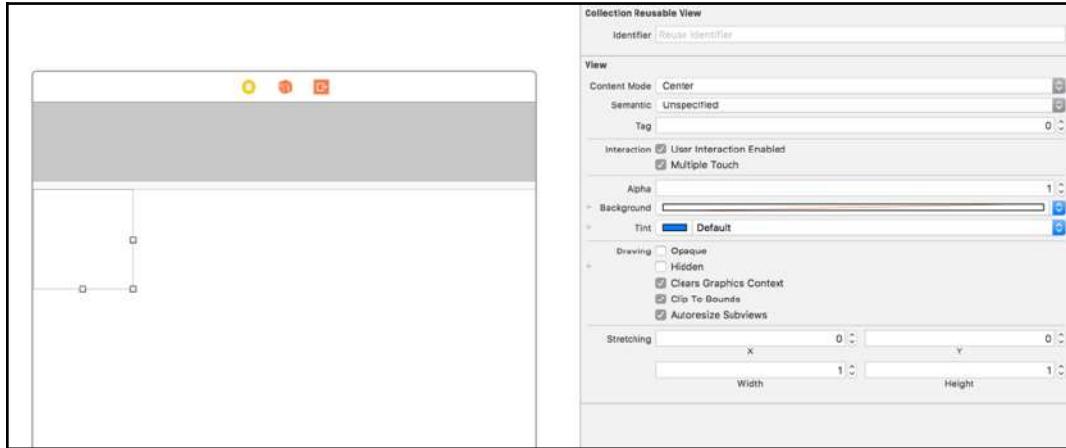


Now that we have a color, we will be able to find our new color in the **Color** dropdown, as you will see next.

Setting up our cell

To set up our cell, we need to perform the following steps:

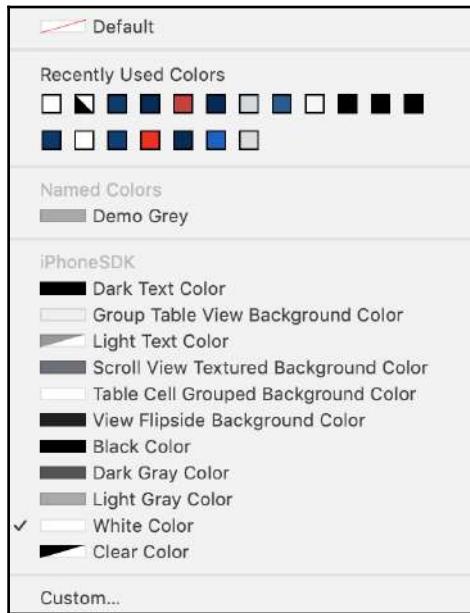
1. In `Main.storyboard`, select the **Collection View** prototype cell, which is the small box inside of your **Collection View**.
2. Open the Attributes inspector in the Utilities Panel:



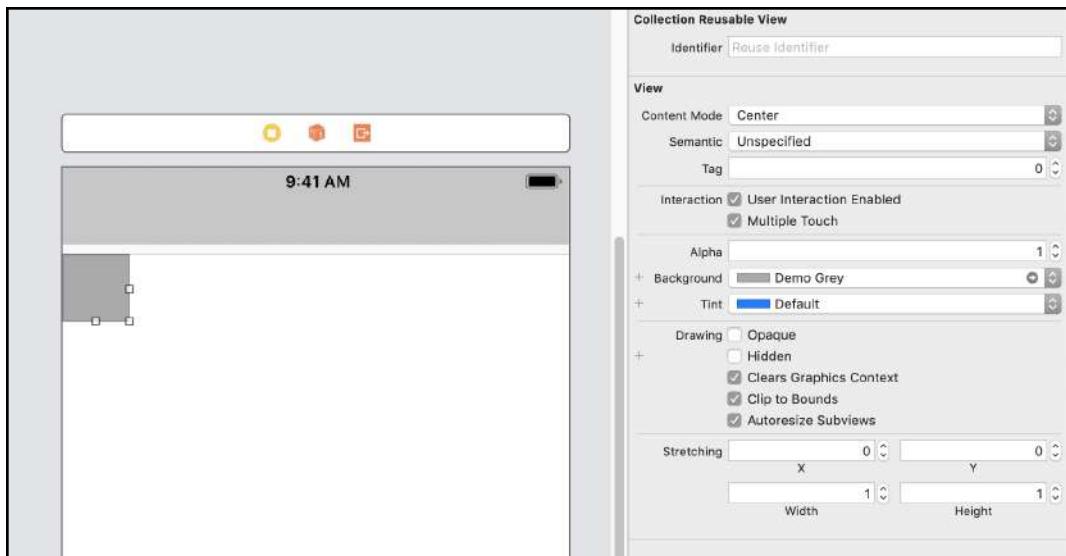
3. Update the following:

- **Identifier:** `exploreCell`
- **Background:** `Demo Grey`

4. To update the background, you will need to click on the drop-down arrow under **Background**. You will see that our **Demo Grey** is added:



You should now see the following:

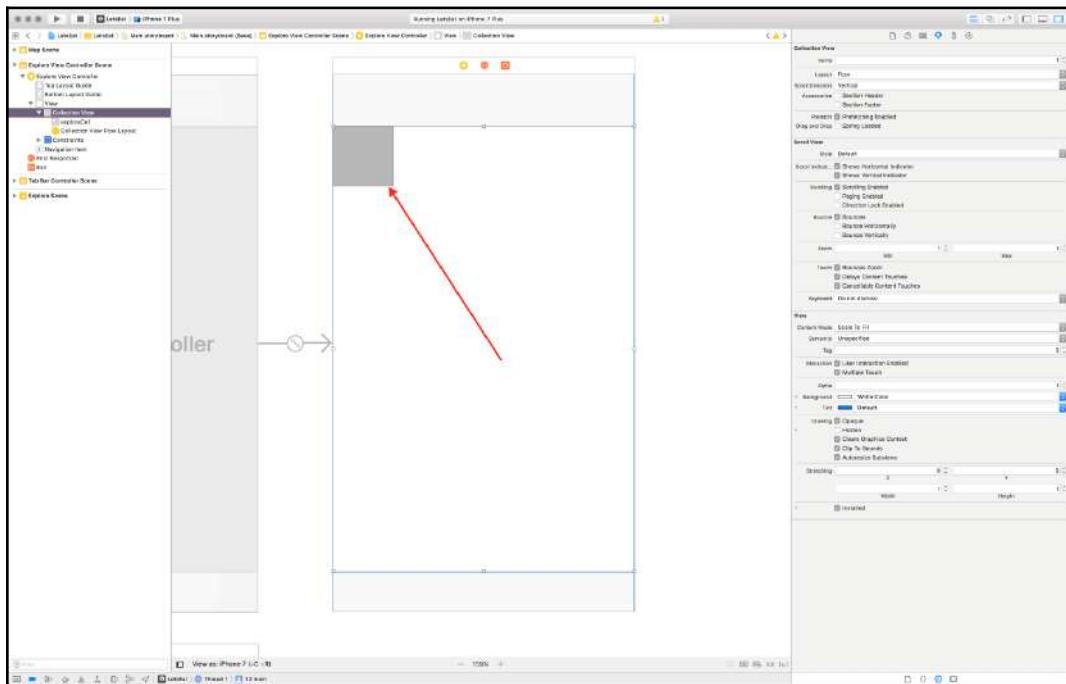


Next, we need to add a section header.

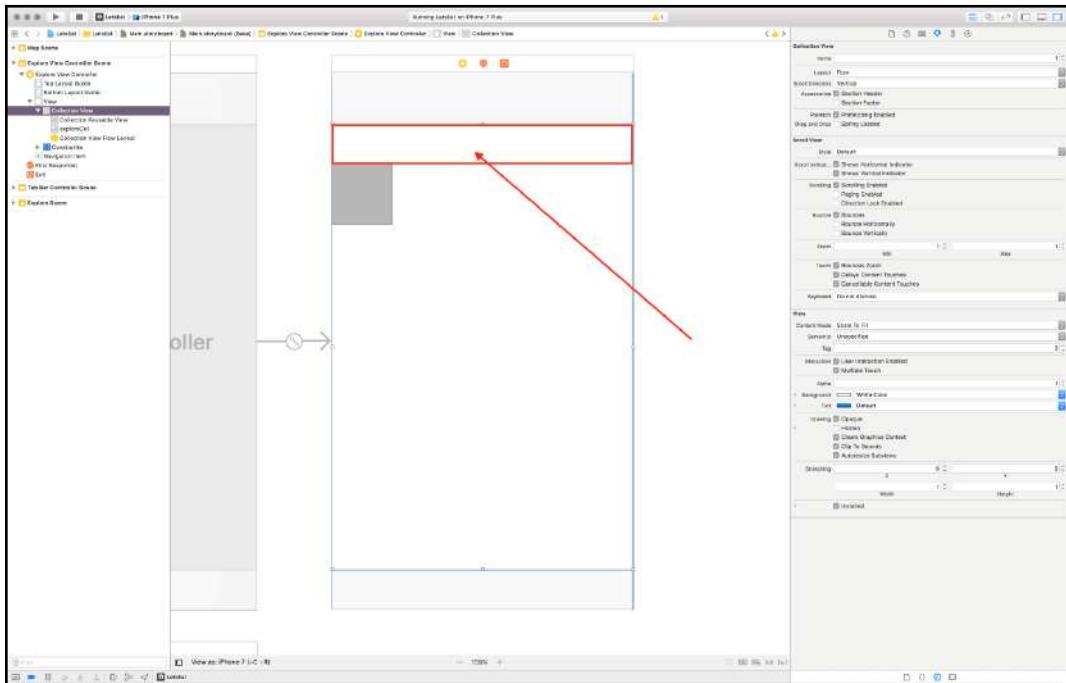
Section header

Our section header will include the page title, the selected location, and a button that we will use to see the locations:

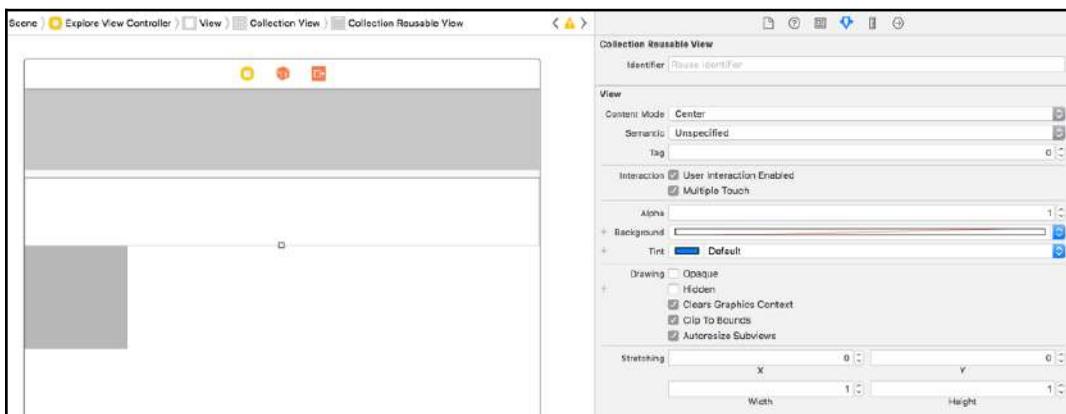
1. Select the **Collection View** in the `Main.storyboard` outline.
2. In your Utilities Panel, select the Attributes inspector and, under **Collection View Accessories**, select the checkbox next to **section header**:



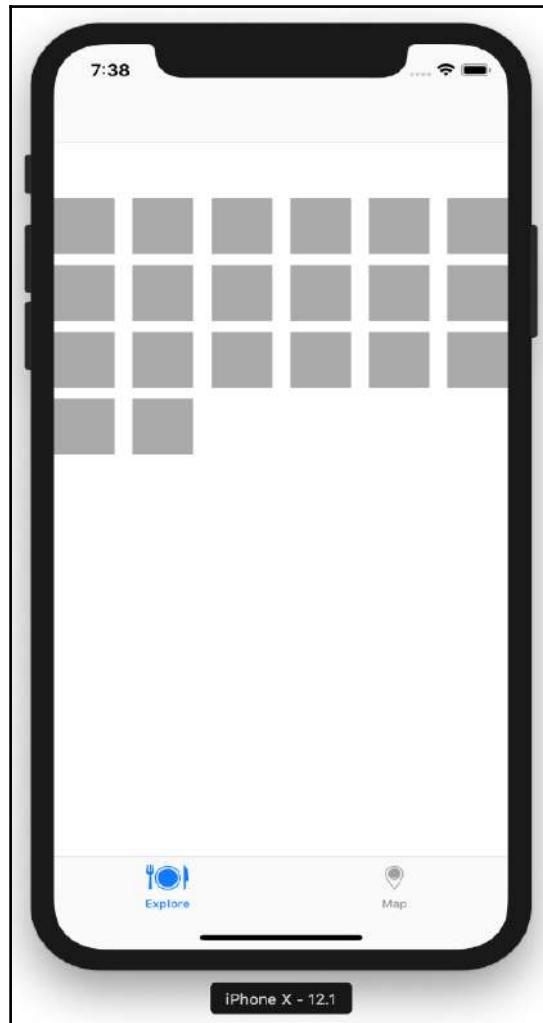
3. You will see a box appear above our **Demo Grey** cell, which is our new section header—select it:



4. In the Attributes inspector in the Utilities Panel, update **Identifier** to **header**:



Let's build and run the project by hitting the Play button (or use *command + R*):



You will see that you now have a grid of boxes and some white space (the section header) near the top of the grid. Before we work on the section header, let's update our grid to match our design of two items per row with a particular cell size.

Updating the grid

To update our grid, we need to take the following steps:

1. Use *command + Shift + O* and, in the **Open Quickly** window, type `Main.storyboard`, and then hit the *Enter* key.
2. Select the **Collection View**, and then, in the Utilities Panel, select **Size Inspector**.
3. Update the following values, based on the simulator that you are currently using. These values may need to be changed so that your grid has two columns of cells, so feel free to alter the values.

For iPhone 7, use the following values:

Fields	Values			
Cell Size	Width: 176	Height: 195		
Min Spacing	For Cells: 0	For Lines: 7		
Section Insets	Top: 7	Bottom: 7	Left: 7	Right: 7

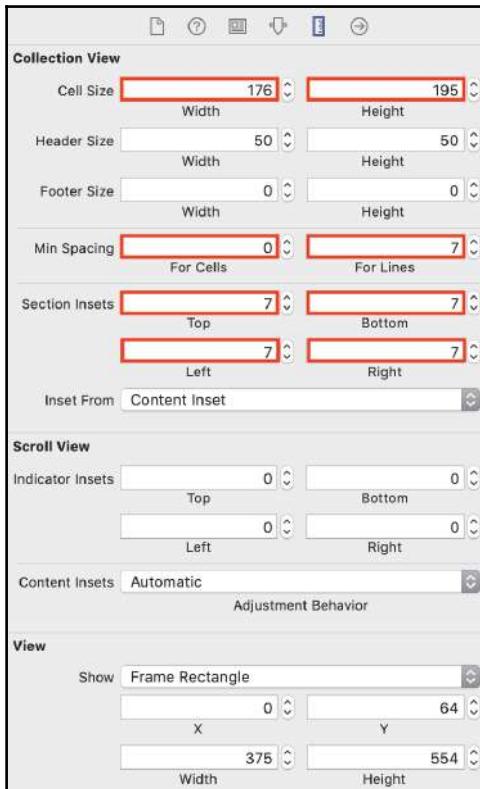
For iPhone 7 Plus, use the following values:

Fields	Values			
Cell Size	Width: 196	Height: 154		
Min Spacing	For Cells: 0	For Lines: 7		
Section Insets	Top: 7	Bottom: 7	Left: 7	Right: 7

For iPhone 4/iPhone SE/iPhone 5/iPhone 5s, use the following values:

Fields	Values		
Cell Size	Width: 150	Height: 154	
Min Spacing	For Cells: 0	For Lines: 7	
Section Insets	Top: 7	Bottom: 7	Left: 7 Right: 7

When you are done, this is what everything should look like:



Challenge: If you are using iPhone XR, or iPhone Xs, try to see whether you can figure out how to make the grid work. For now, as we just did, we will use storyboard settings to get our cells set up. Later in the book, we will make this dynamic so that our widths and heights adjust with code. Next, we will work on our section header.

Adding a modal

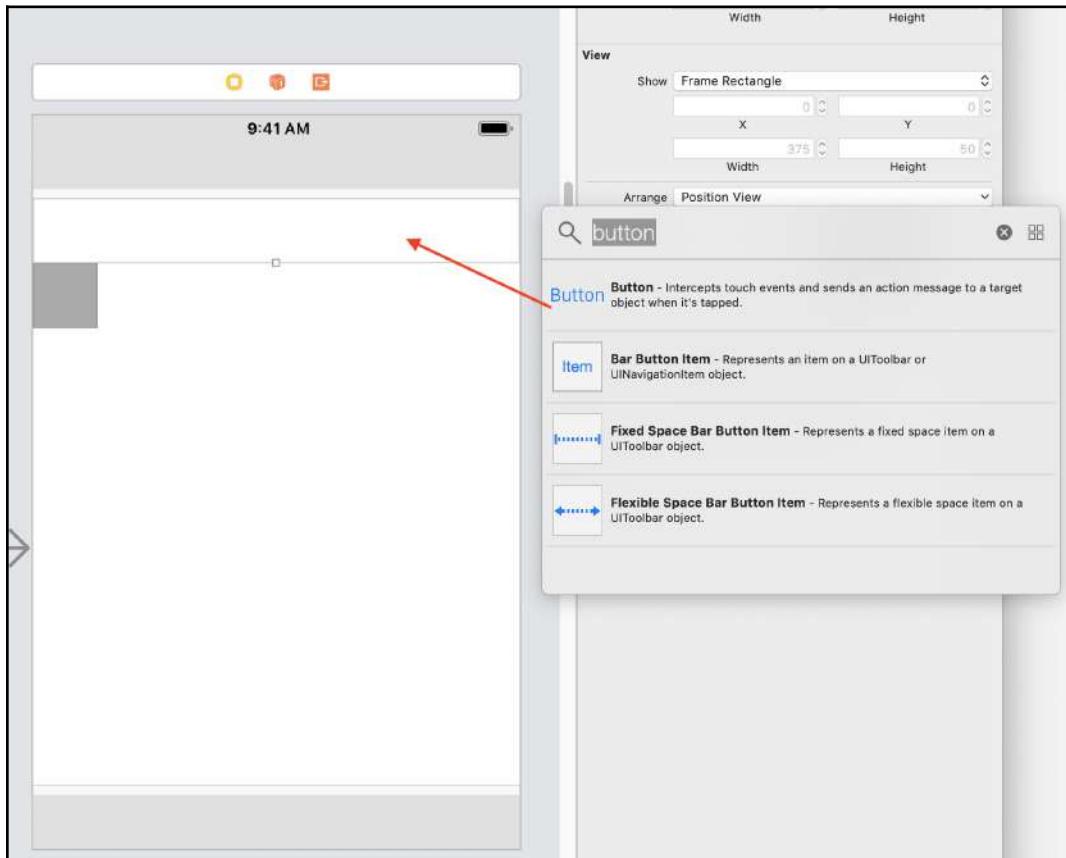
Let's review the design for the section header:



Note that we have a **+ Location** button that will display our locations. Let's add that modal now:

1. While in the `Main.storyboard` file, select the object library and, in the filter field at the bottom of the Library pane, type `button`.

2. Drag and drop the Button component into the section header we created in our **Explore View Controller**:

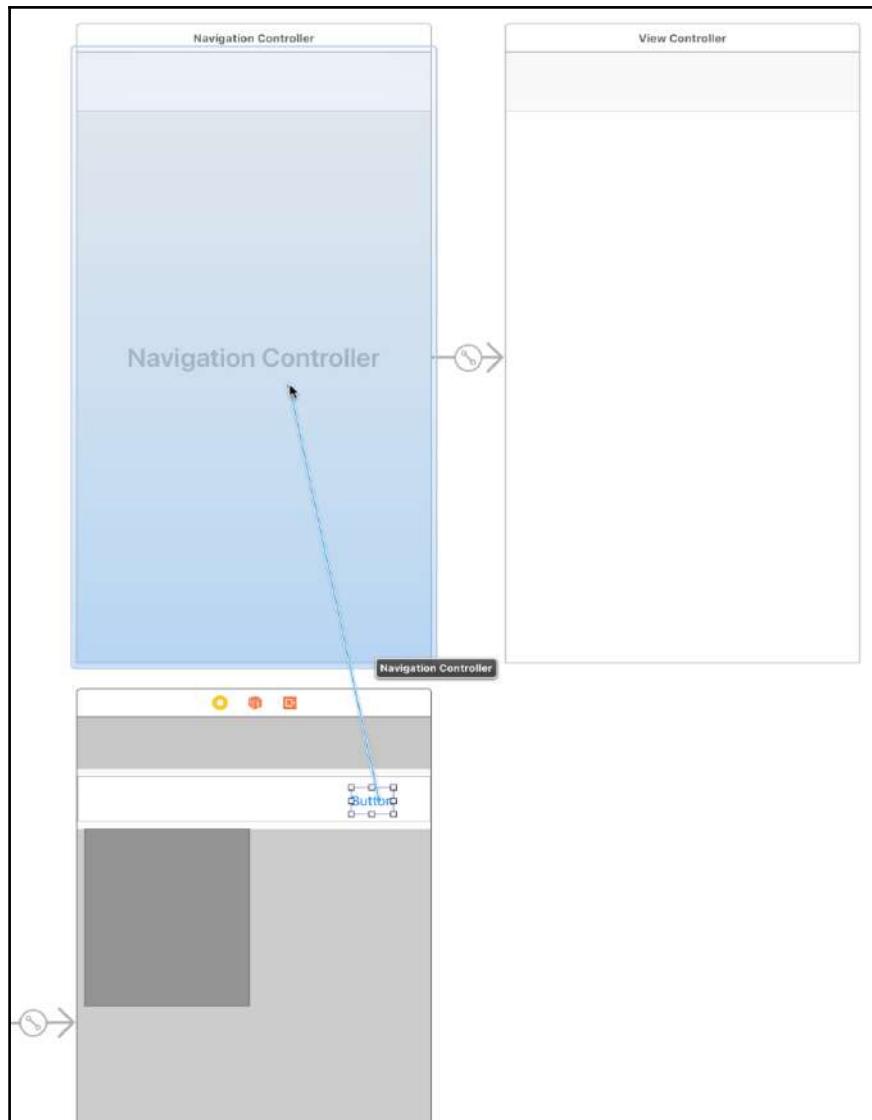


Ignore the layout warning. We will format the button later regarding location and size, and that will get rid of the warning.

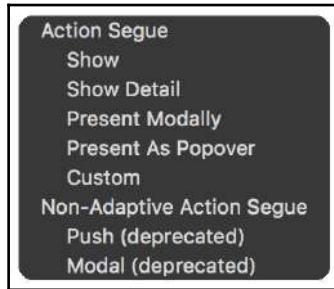
Next, we need to add another View Controller to our storyboard:

1. In the filter, type `viewcontroller`, and drag and drop the `ViewController` component above the **Explore View Controller** in `Main.storyboard`.
2. With the **View Controller** selected, navigate to **Editor | EmbedIn | NavigationController**.

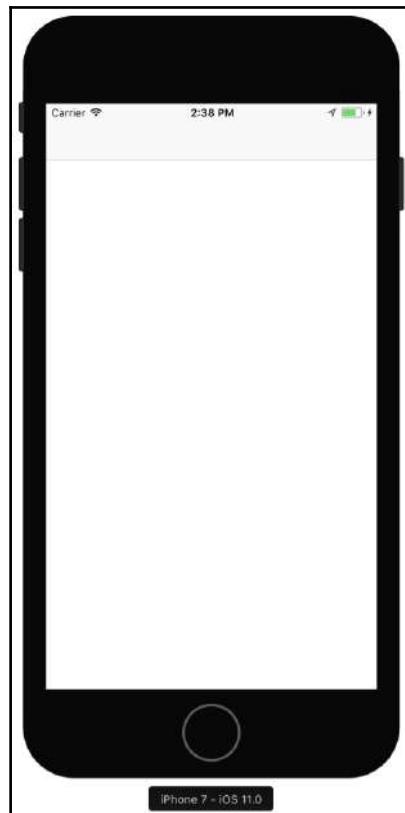
3. *Control + drag* from where it says **Button** in the View Controller, under the **Explore** tab, to the Navigation Controller that was just created (you can also do this within Outline view, by *Control + dragging* from the button to the new Navigation Controller you just created):



- When you let go, you will be presented with the following menu, and you should select **Present Modally**:

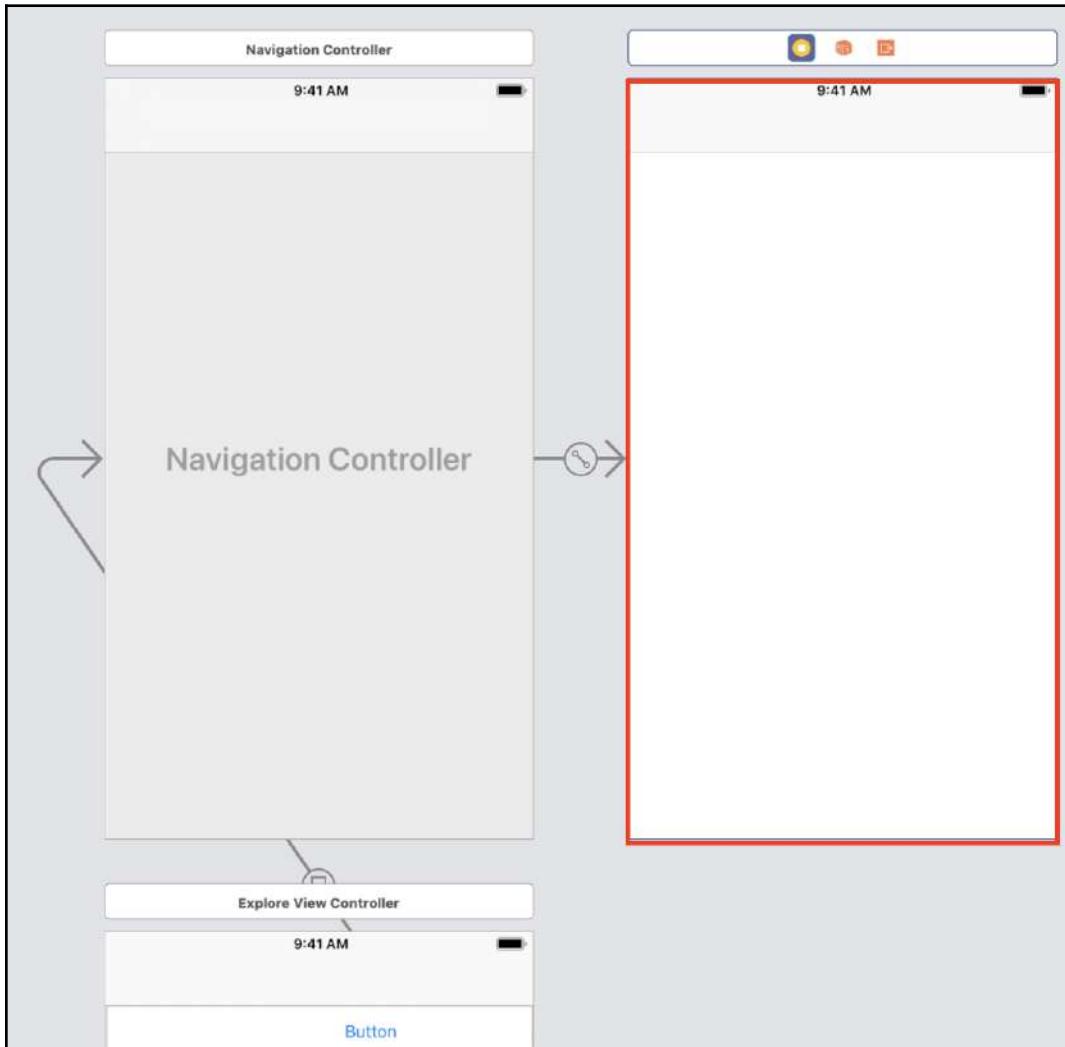


Now, let's run the project by hitting the Play button (or use *command + R*). You will see that our button now launches a modal. In the next chapter, we will make this button match our design:

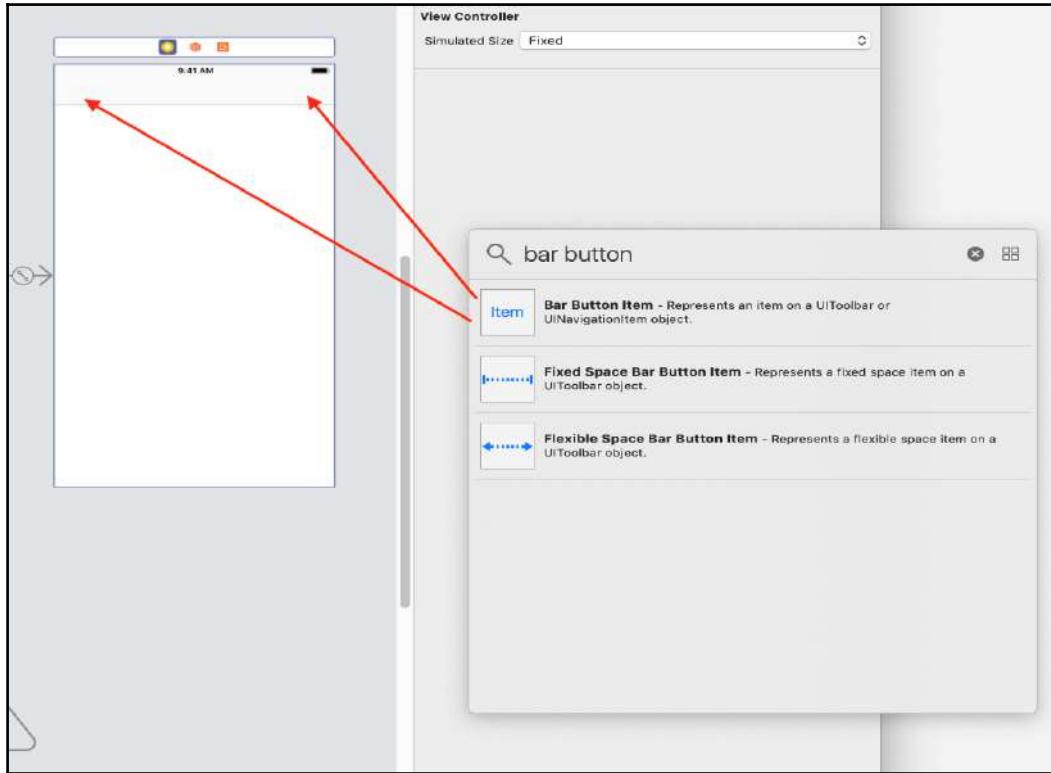


Currently, as you can see in the preceding screenshot, we cannot dismiss this modal. Therefore, we need a cancel button and a done button to dismiss the view. Let's fix this:

1. Open `Main.storyboard` and then go to your **View Controller** (not the **Navigation Controller**) of your modal:



2. Open the object library (*command + Shift + L*) and type **bar button** into the filter area of the objects library in the Utilities Panel.
3. Drag and drop a **Bar Button Item** into the right-hand side of the **Navigation Bar** of your View Controller Scene:



4. Drag another **Bar Button Item** into the left-hand side of the **Navigation Bar**.

5. You should have two **Bar Button Items** that both say **Item**:

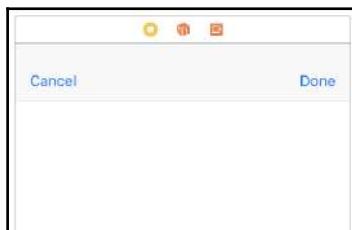


Updating Bar Button Items

Next, we need to update both of the **Bar Button Items** to say cancel and done:

1. Select the left-hand **Bar Button Item**, and, in the Utilities Panel, select the Attributes inspector.
2. Click on **System Item** and select **Cancel** in the drop-down menu.
3. Select the right-hand **Bar Button Item**, and, while still in the Attributes inspector in the Utilities panel, update **System Item** to **Done**.

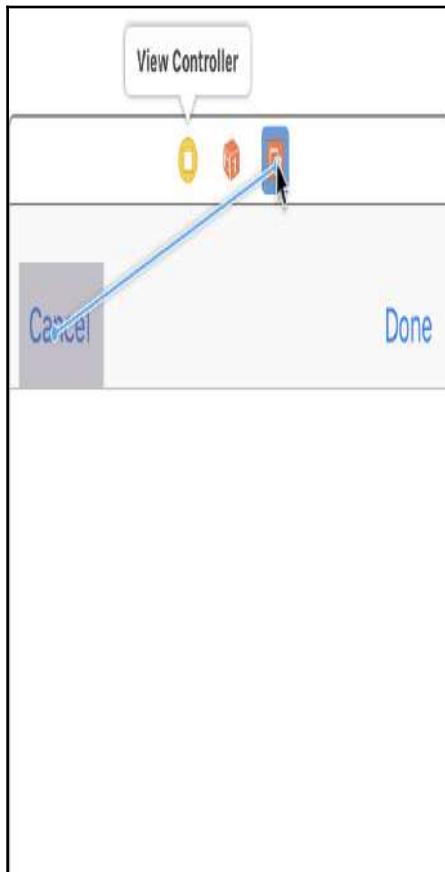
Now, you should see **Cancel** on the left and **Done** on the right:



Unwinding our Cancel button

Now that we have our buttons, we want to dismiss the modal when a user hits **Cancel**.

In the `Main.storyboard`, *Control + drag* from the **Cancel** button to **Exit**:



You can also do this in the Outline view.

You will see a window popup that says **Action Segue** and **unwindLocationCancelWithSegue**. Select `unwindLocationCancelWithSegue`:

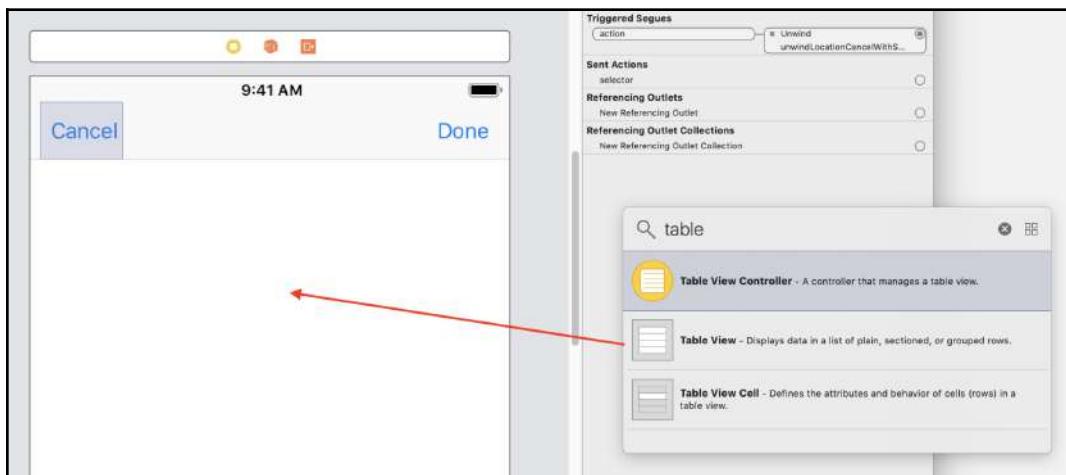
Action Segue
`unwindLocationCancelWithSegue:`

Let's build and run the project by hitting the Play button (or use *command +R*), and test our **Cancel** button. It should now dismiss the View. We will update the **Done** button when we add code later.

Adding our first Table View

Now, let's add a `UITableView` into our `UIViewController`:

1. In the Utilities Panel of `Main.storyboard`, in the filter field, type `table`, then drag the Table View onto the scene:



2. Select the Pin icon and enter the following values:
 - Set all values under **Add New Constraints** to 0.
 - The **Constrain to margins** checkbox should be unchecked.
3. Click on **Add 4 Constraints**.

If you build and run the project, and then launch the modal, you will see an empty Table View. We will complete this Table View later.

Summary

We are about halfway through the setup of our UI structure. In this chapter, we created our **Collection View** with a dummy cell. Dummy cells allow us to continue to work on the basic structure of our app and focus on the design of the app, getting all of the assets ready to go before we add code. We also added our first prototype header, as well as presenting a modal to the user.

In the next chapter, we will complete the rest of our basic structure, before concentrating on adapting our structure to match our design.

9

Finishing Up Our App Structure in Storyboard

The more we do storyboard work, the easier it gets. I remember that, when I started Xcode, it was a bit overwhelming because of all the panels, and it took me time to get comfortable. Any time I speak with someone looking to get into iOS, I always tell them to dedicate at least 10-15 minutes a day to it for the first six months. It seems like a lot, but it makes a difference when you are trying to learn. If you step away for a week and try to come back, it's like starting from square one; at least, it was for me. When I finally started to catch on was when I was in Xcode every day and was relentless.

In the previous chapter, we got our **Explore** and **Location** both set up. In this chapter, we are still working on just the structure, and in the next couple of chapters we will work on the design.

We will cover the following in this chapter:

- Restaurant View Controller
- Restaurant Detail View Controller
- Reviews View Controller
- The Map tab

Adding our Restaurant List View

Our restaurant list has the same basic setup as in the previous chapter. If you think you have a grasp of this, now is an excellent time to challenge yourself. If you think you still need more practice, keep reading and let's set up the restaurant list:

1. Select the `Main.storyboard` file, making sure that you are zoomed out and can see all of your scenes (depending on your screen resolution). Open the object library (`command + shift + L`).
2. Drag out a View Controller—it should be the first item in the list—put it next to Explore View Controller.
3. Open the object library (`Command + Shift + L`) and, in the filter field, type `collectionview`.
4. Click on and drag **Collection View** and drop it onto the new **View Controller** we just added, next to the Explore View Controller.
5. Select the **Pin** icon and enter the following values:
 - All values under **Add New Constraints** are set to 0.
 - Make sure to uncheck **Constrain to margins**.
6. Click on **Add 4 Constraints**.

We now have our **Collection View** component set up for our Restaurant list.

Hooking up our outlets

Let's now link our file, `RestaurantViewController`, to our new `UIViewController` in the storyboard:

1. Select the `UIViewController` with the **Collection View** that we just created.
2. In the Utility panel, select the Identity inspector. Under **Custom Class**, in the **Class** drop-down menu, select `RestaurantViewController` and hit the *Enter* key.
3. Select the **Connections Inspector** in the Utilities Panel.
4. Under **Outlets**, (just like we did earlier) click on the `collectionView` circle and drag from the circle to the **Collection View** that we just added inside of your `UIViewController`.

Now that we have our **Collection View** hooked up, we need to hook up the data source and delegate. The data source and delegate allow us to pass data to our **Collection View** as well as to know when our **Collection View** has some interaction. Let's do that now by doing the following:

1. In your scene, select your **Collection View**. In your Utilities Panel, select the **Connections Inspector**.
2. Click on and drag the `dataSource` property to the **Restaurant View Controller** in your Outline view.
3. Click on and drag the `delegate` property to the **Restaurant View Controller** in your Outline view.

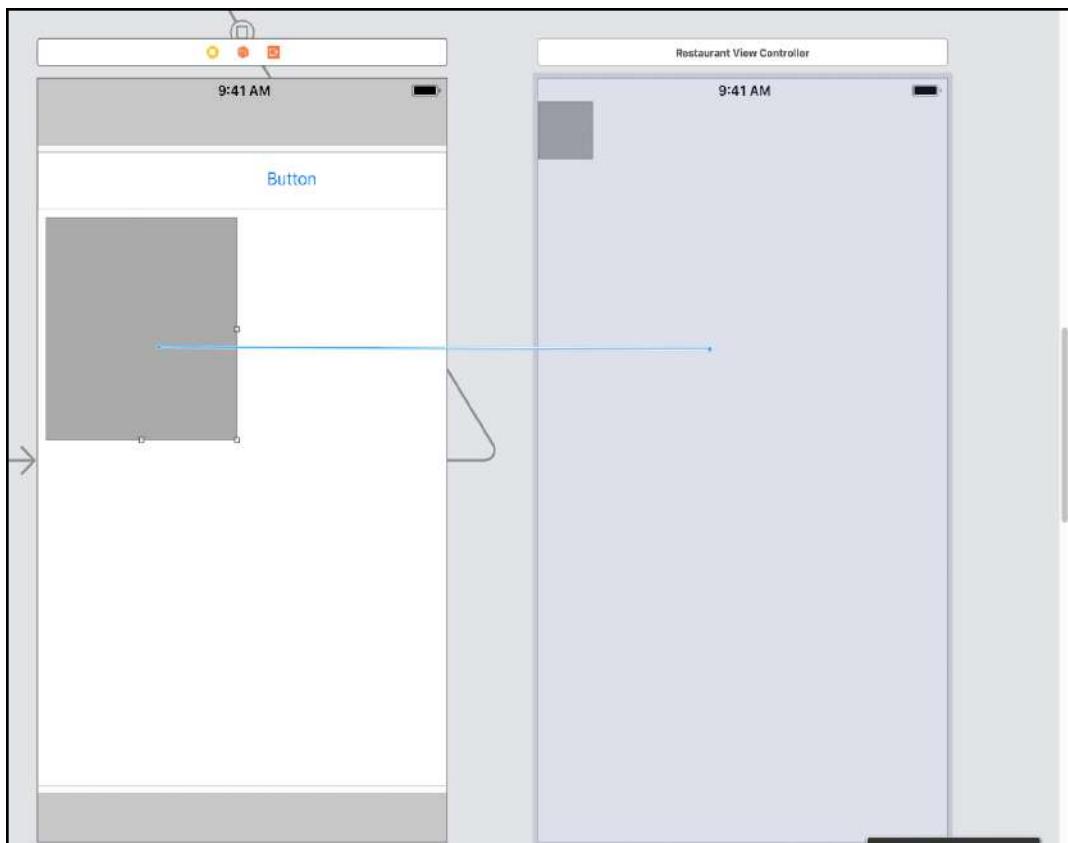
Finally, let's set up our cell to have a color.

Setting up our cell

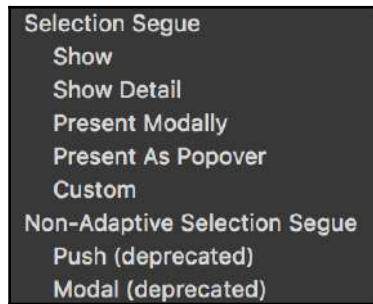
In `Main.storyboard`, select the small box inside of your **Collection View**. The small box is your **Collection View** prototype cell:

1. Open the Attributes inspector in the Utilities Panel.
2. Update the following:
 - **Identifier:** `restaurantCell`
 - **Background:** Demo Grey

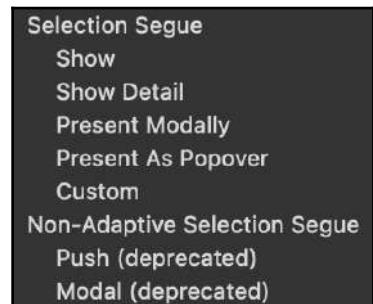
3. Control + drag from the explore cell to Restaurant View Controller:



- When you let go, you are presented with the following menu—select **Show**:



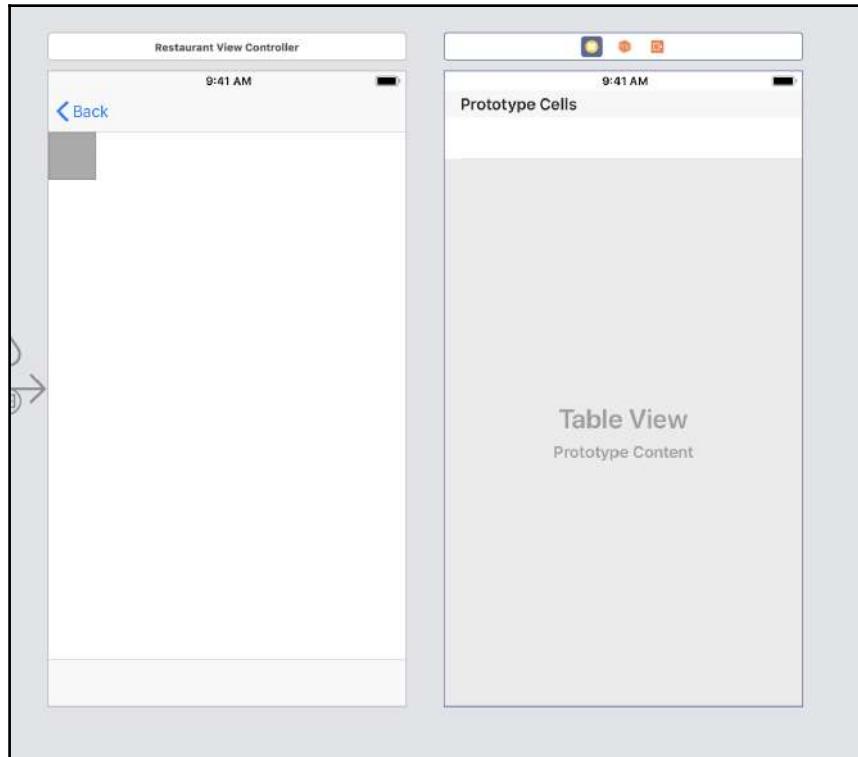
Now, let's run the project by hitting the Play button (or using *Command + R*). You will now be able to tap on an explore cell and see the following:



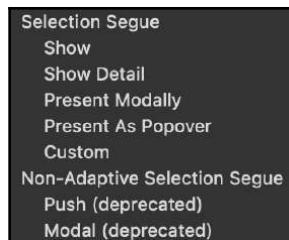
Next, we want the user to be presented with the restaurant's details when they touch a restaurant. We will use a static Table View Controller to do our detail. Using a static Table View allows us to create content without code. We will still have to hook up our data but, in the upcoming chapters, you will see how static Table Views come in handy. Let's set up the restaurant details:

- Select the `Main.storyboard` file, making sure that you are zoomed out and can see all of your scenes (depending on your screen resolution). In the Utilities Panel, ensure that you have the object library tab selected.

2. In the filter field, type `tableviewcontroller` (make sure it's the controller—it will have a yellow icon). Drag this **Table View Controller** and put it next to **Restaurant View Controller**:



3. *Control-drag* from the restaurant cell button to the **Restaurant Detail Table View Controller**. When you let go, you are presented with the following menu, and you should select **Show**:



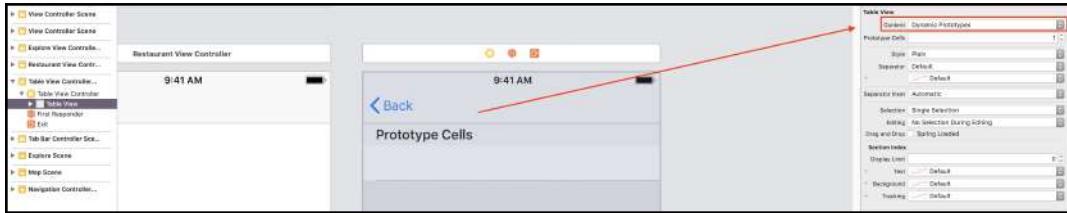
4. Click on the **Table View** inside of the Outline:

```

2018-11-20 10:25:11.339040-0500 LetsEat[59237:10190392] *** Assertion failure in -[UICollectionView
_dequeueReusableViewOfKind:withIdentifier:forIndexPath:viewCategory:], /BuildRoot/Library/Caches/
com.apple.xbs/Sources/UIKitCore_Sim/UIKit-3698.93.8/UICollectionView.m:5372
2018-11-20 10:25:11.344351-0500 LetsEat[59237:10190392] *** Terminating app due to uncaught exception
'NSInternalInconsistencyException', reason: 'could not dequeue a view of kind: UICollectionViewElementKindCell
with identifier restaurantCell - must register a nib or a class for the identifier or connect a prototype
cell in a storyboard'
*** First throw call stack:
(
    0  CoreFoundation                      0x00000001056541bb __exceptionPreprocess + 331
    1  libobjc.A.dylib                     0x0000000103c9735 objc_exception_throw + 48
    2  CoreFoundation                      0x0000000105653f42 +[NSErrorException raise:format:arguments:] + 98
    3  Foundation                          0x000000010369a877 -[NSAssertionHandler
handleFailureInMethod:object:file:lineNumber:description:] + 194
    4  UIKitCore                           0x0000000107847706 -[UICollectionView
_dequeueReusableViewOfKind:withIdentifier:forIndexPath:viewCategory:] + 2536
    5  UIKitCore                           0x0000000107847991 -[UICollectionView
dequeueReusableCellWithReuseIdentifier:forIndexPath:] + 169
    6  LetsEat                            0x00000001033697db
$S7LetsEat24RestaurantViewControllerC010collectionD0_13cellForItemAtSo012UICollectionViewD4CellCSo0kD0C_10F
oundation9IndexPathVtF + 171
    7  LetsEat                            0x000000010336987c
$S7LetsEat24RestaurantViewControllerC010collectionD0_13cellForItemAtSo012UICollectionViewD4CellCSo0kD0C_10F
oundation9IndexPathVtFTo + 108
    8  UIKitCore                           0x00000001078312d8 -[UICollectionView
_createPreparedCellForItemAtIndexPath:withLayoutAttributes:applyAttributes:isFocused:notify:] + 314
    9  UIKitCore                           0x0000000107831198 -[UICollectionView
_createPreparedCellForItemAtIndexPath:withLayoutAttributes:applyAttributes:] + 31
    10 UIKitCore                          0x000000010783684f -[UICollectionView _updateVisibleCellsNow:] +
6164
    11 UIKitCore                          0x000000010783c076 -[UICollectionView layoutSubviews] + 364
    12 UIKitCore                          0x00000001084c9795 -[UIView(CALayerDelegate)
layoutSublayersOfLayer:] + 1441
    13 QuartzCore                         0x0000000109a51b19 -[CALayer layoutSublayers] + 175
    14 QuartzCore                         0x0000000109a569d3
_ZN2CA5Layer16layout_if_neededEPNS_11TransactionE + 395
    15 QuartzCore                         0x00000001099cf7ca
_ZN2CA7Context18commit_transactionEPNS_11TransactionE + 342
    16 QuartzCore                         0x0000000109a0697e _ZN2CA11Transaction6commitEv + 576
    17 UIKitCore                          0x0000000107fd9701 _UIApplicationFlushRunLoopCATransactionIfTooLate
+ 165
    18 UIKitCore                          0x00000001080d3569 __handleEventQueueInternal + 6874
    19 CoreFoundation                     0x00000001055b9721
__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__ + 17
    20 CoreFoundation                     0x00000001055b8f93 __CFRunLoopDoSources0 + 243
    21 CoreFoundation                     0x00000001055b363f __CFRunLoopRun + 1263
    22 CoreFoundation                     0x00000001055b2e11 CFRunLoopRunSpecific + 625
    23 GraphicsServices                  0x000000010d8551dd GSEventRunModal + 62
    24 UIKitCore                          0x0000000107fdf81d UIApplicationMain + 140
    25 LetsEat                           0x000000010336a927 main + 71
    26 libdyld.dylib                     0x0000000106aed575 start + 1
)
libc++abi.dylib: terminating with uncaught exception of type NSErrorException
(lldb)

```

5. Make sure that you have the **Attributes inspector** opened in the Utilities Panel, then change the **Table View** content from **Dynamic Prototypes** to **Static Cells**:



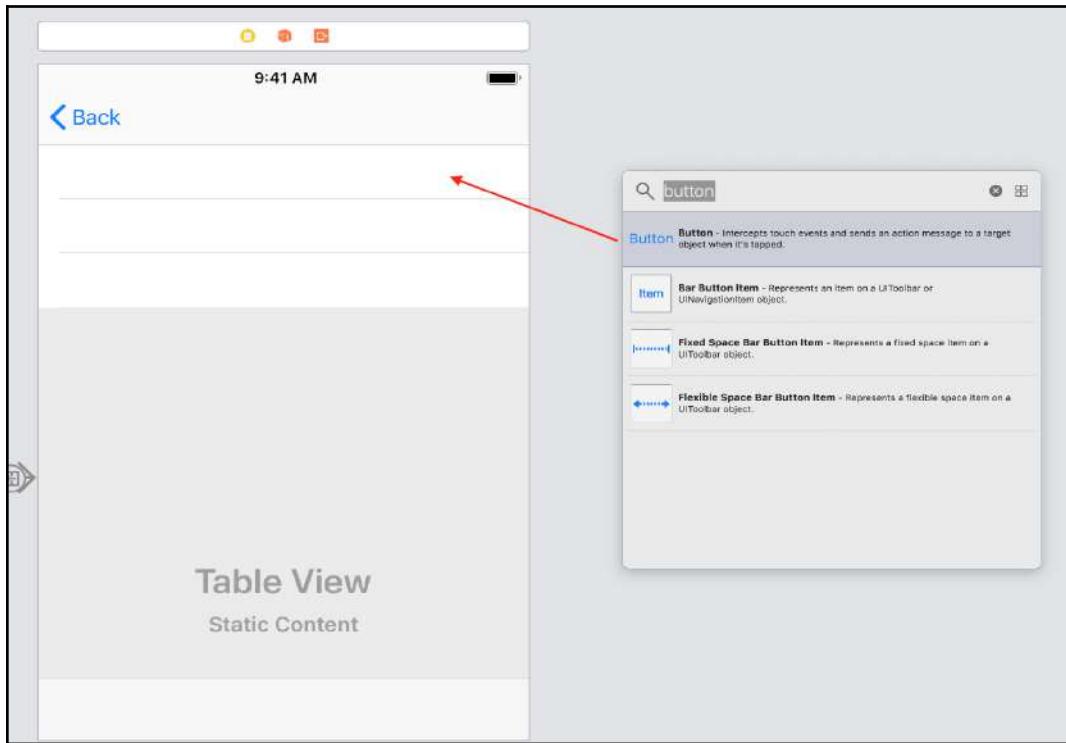
Now, let's run the project by hitting the Play button (or using *Command + R*). You will now be able to tap on a restaurant cell and see the following:



Adding the Reviews View

We have our static Table View set up now; we need another view that allows us to view restaurant reviews. Let's add that now:

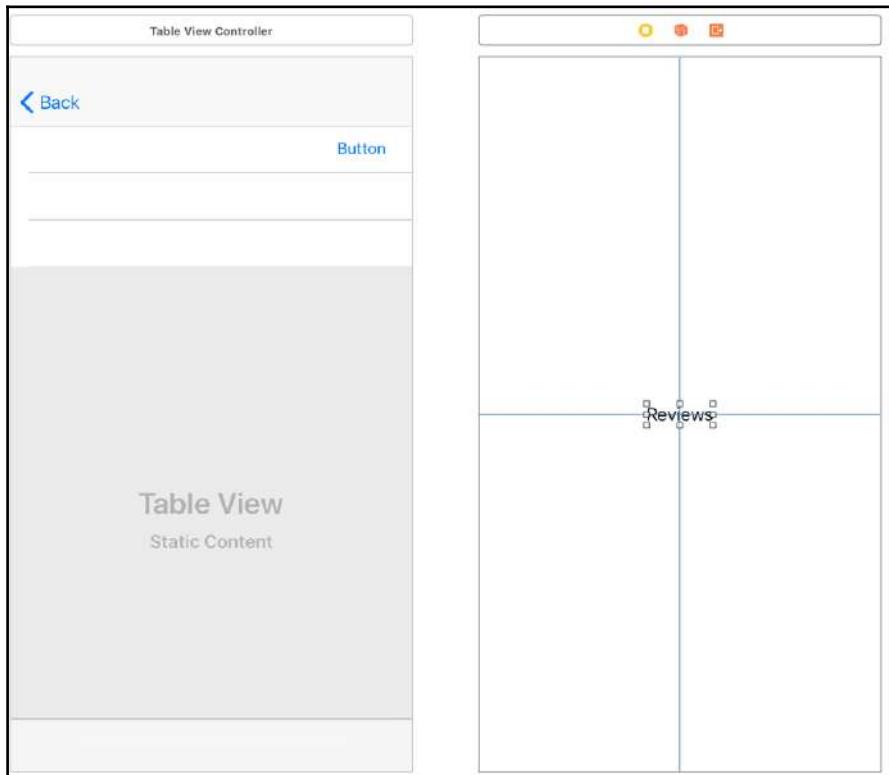
1. In the object library, in the filter field, type button. Drag this button and put it next to one of the `tableview` cells:



2. Select the `Main.storyboard` file, making sure that you are zoomed out and can see all of your scenes (depending on your screen resolution). In the Utilities panel, ensure that you have the object library tab selected.

3. In the filter field, type `viewcontroller`. Drag this **View Controller** and put it next to the **Restaurant Detail View Controller**.
4. In the filter field, type `label`.
5. Click on and drag **Label** and drop it onto the new **View Controller** we just added, next to the **Restaurant Detail View Controller**.
6. Double-click in the **Label** and add the `Reviews` text.
7. Select the **Align** icon that is to the left of the **Pin** icon and check the following boxes that appear:
 - Horizontally in container
 - Vertically in container
8. Click on **Add 2 Constraints**.

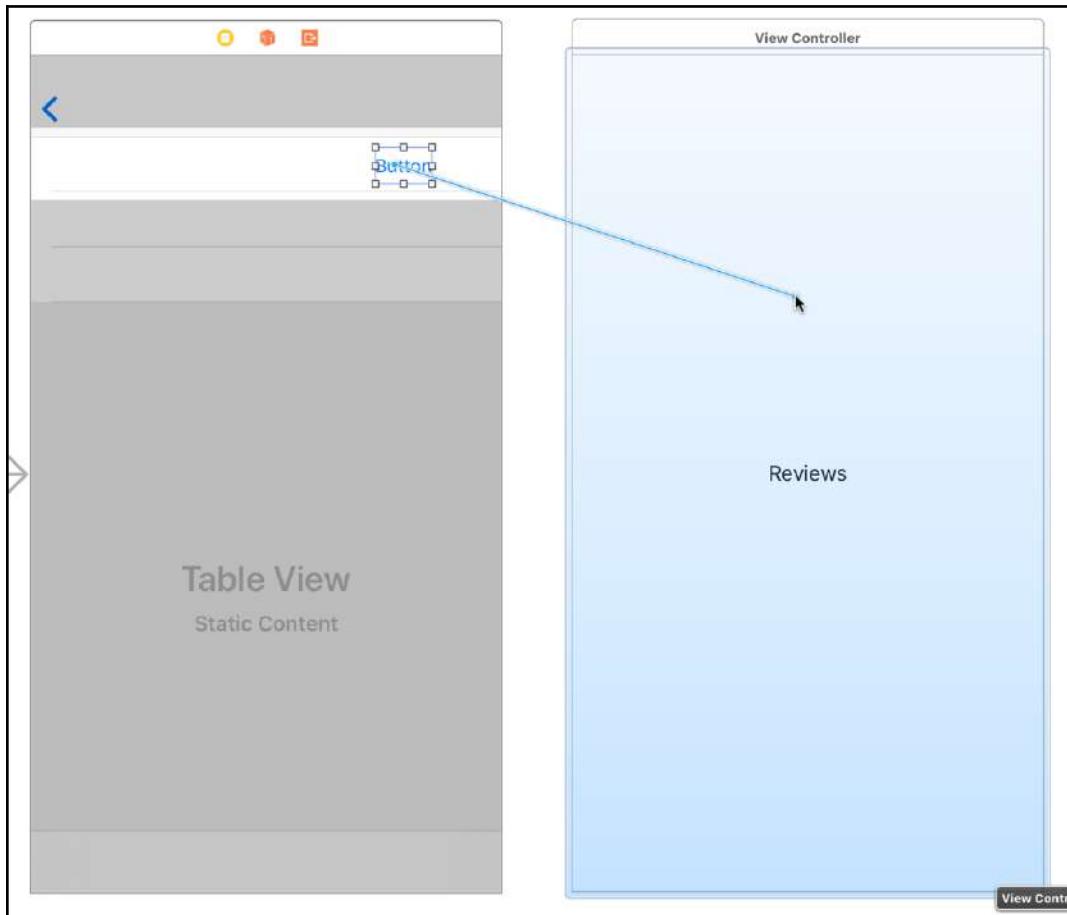
When you are done, you will see the following:



Viewing reviews

Now, we need to add a segue to be able to get to the Reviews View Controller:

1. *Control + drag* from the button to the **Review View Controller** that we added earlier.
2. When you let go, you are presented with a menu, and you should select **Show**:

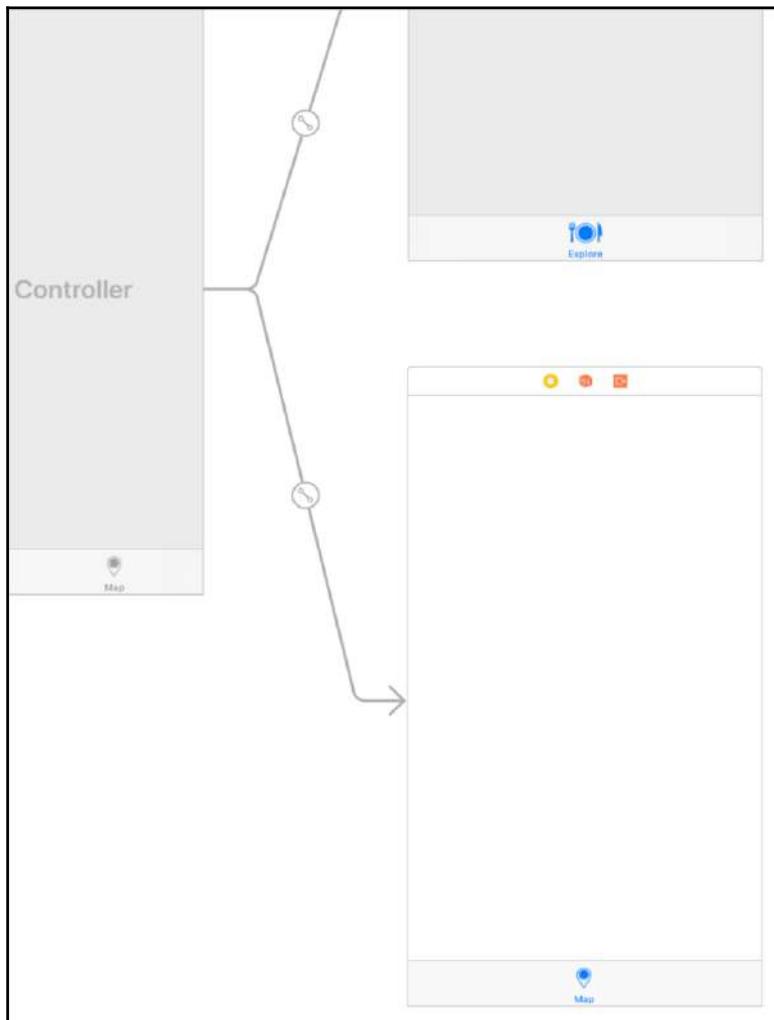


Let's run the project by hitting the Play button (or using *Command + R*). You will now be able to tap on the button in restaurant details and see the following:



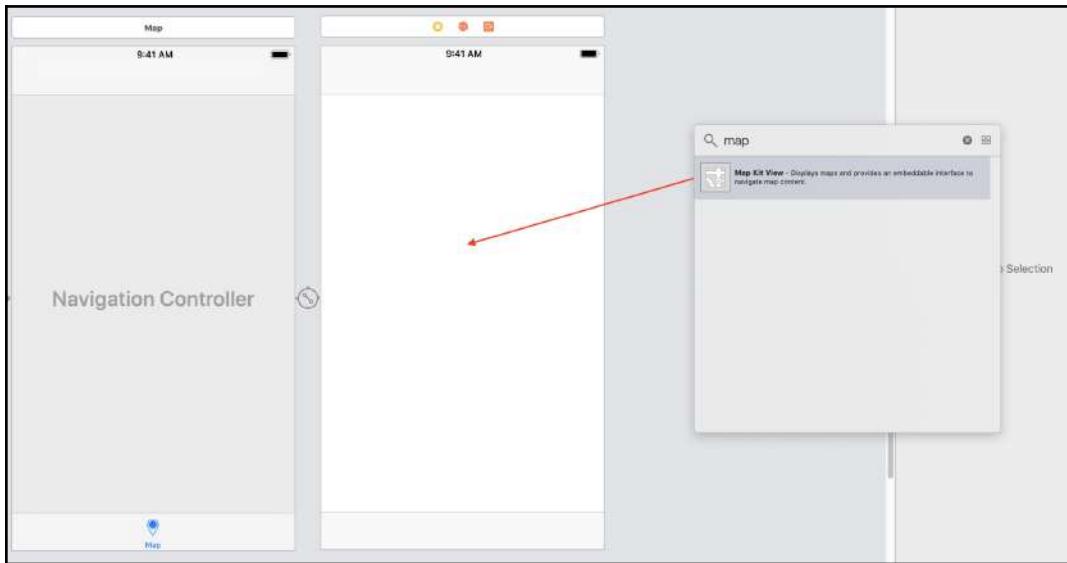
Map Kit View

The last thing we need to do is to set up our **Map** tab. Select the `Main.storyboard` file and find the **View Controller** connected to the **Map** tab:



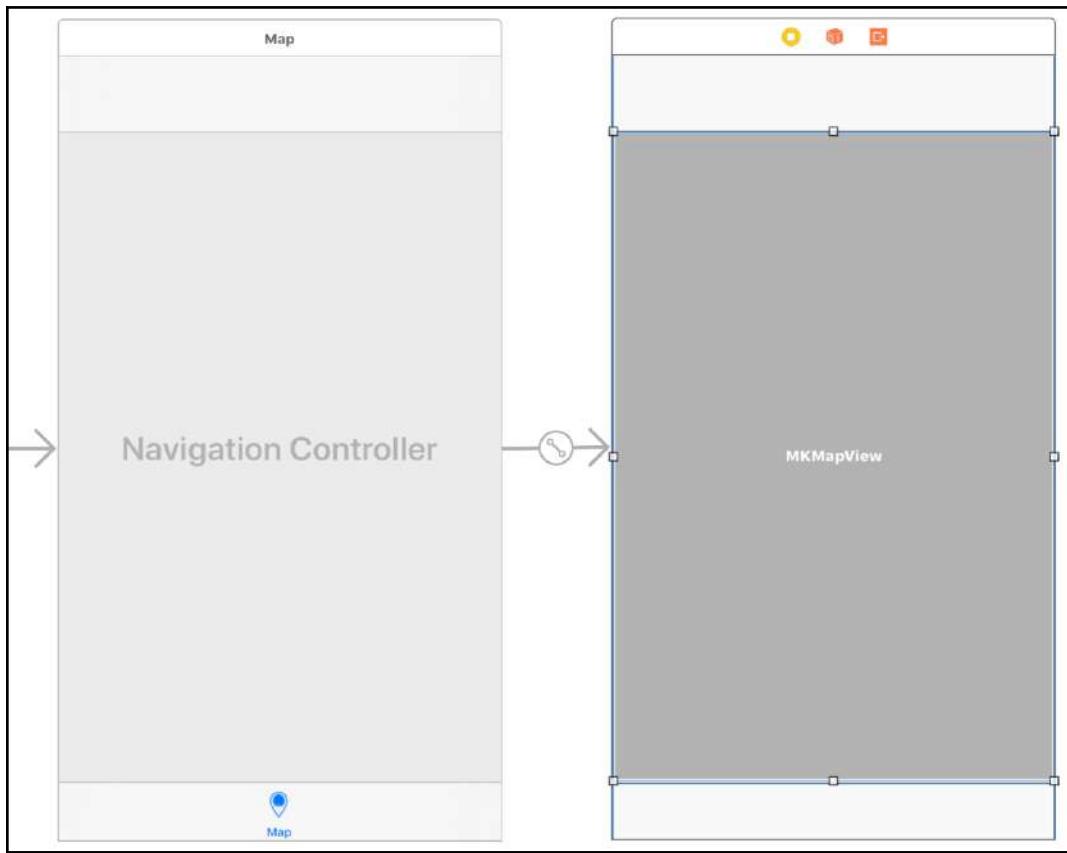
Let's get started:

1. Open the Object Library (*Command + shift + L*), then type map.
2. Drag and drop **Map Kit View** onto the **Map View Controller**:

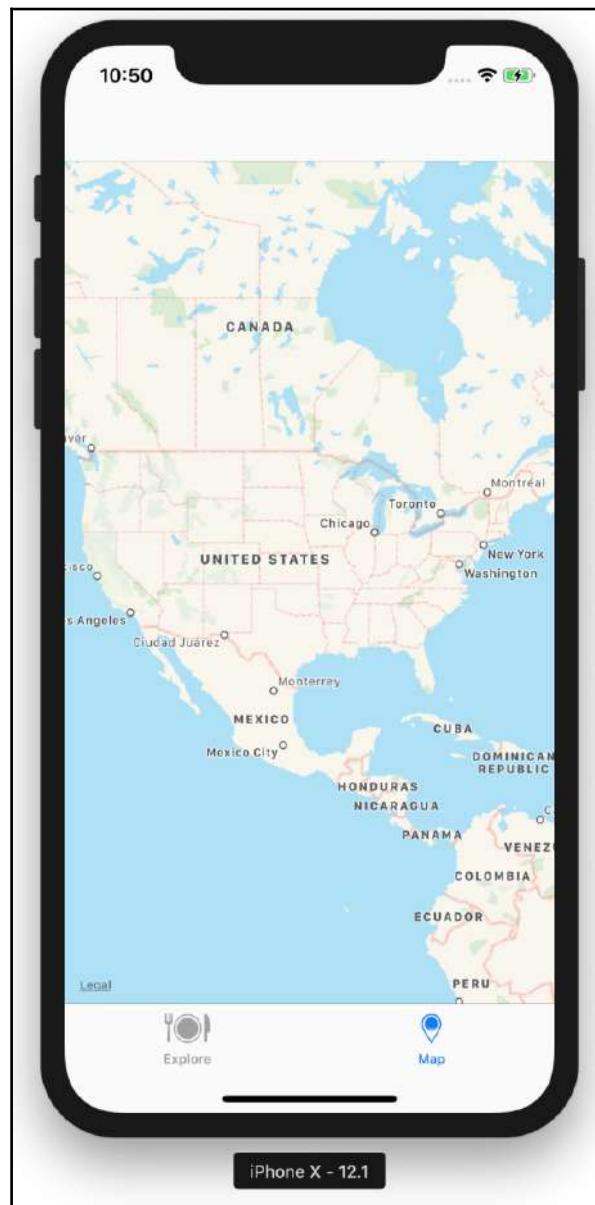


3. Select the **Pin** icon and enter the following values:
 - All values under **Add New Constraints** are set to 0.
 - Uncheck the **Constrain to margins** checkbox.
 - Click on **Add 4 Constraints**.

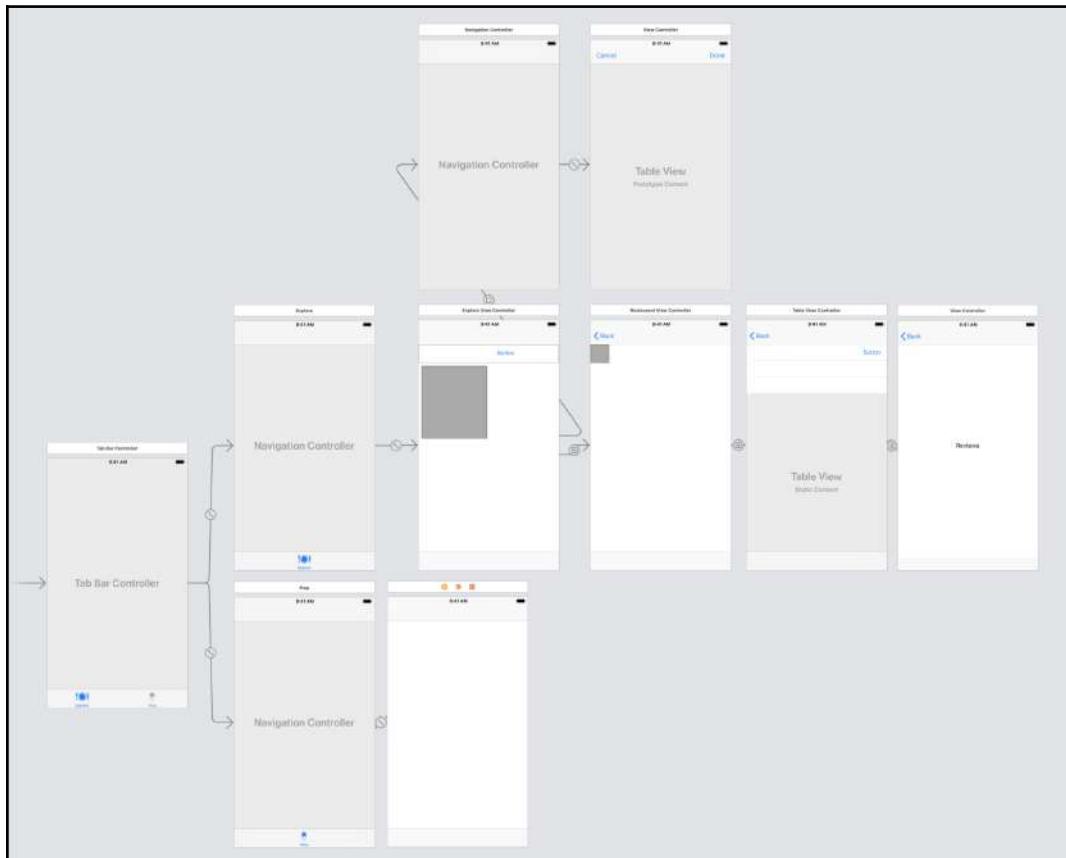
4. Your **View Controller** should look like the following when you are done:



5. Run the project by hitting the Play button (or using *Command + R*) and selecting the **Map** tab:



We now have both tabs set up, but, as we progress through the book, we will add more scenes to the storyboard. The following is what your `Main.storyboard` file should look like:



Summary

In this chapter, we finished the application structure. We hooked up our explore cell to a restaurant list. Then, we were able to connect a restaurant to a detail. Next, we added a button to our details, which allows us to see restaurant reviews. Lastly, we added a map to our **Map** tab.

At this point, a good challenge would be to see whether you can get back to this point. Try starting from when we created the project toward the end of [Chapter 6, Starting the UI Setup](#), in the *Creating a new project* section. See whether you can get from that point to here without the book and without missing anything. This will help you, and it is something I like to do with those I mentor.

In the next chapter, we will start digging more into the design and getting our app to look like the design visually.

10

Designing Cells

In this chapter and throughout this book, we will adjust our app to match the design we reviewed earlier. However, the specifics of the design, such as custom fonts, are there as examples; you should feel free to change things to match your taste. By experimenting while learning, you should get a better understanding of how things work and become more comfortable using Xcode. I would recommend that you first thoroughly understand the lessons before experimenting; however, I highly encourage you to have fun and make the app your own.

In this chapter, we will be working with the following:

- Table View Cells
- Collection View Cells
- Auto Layout

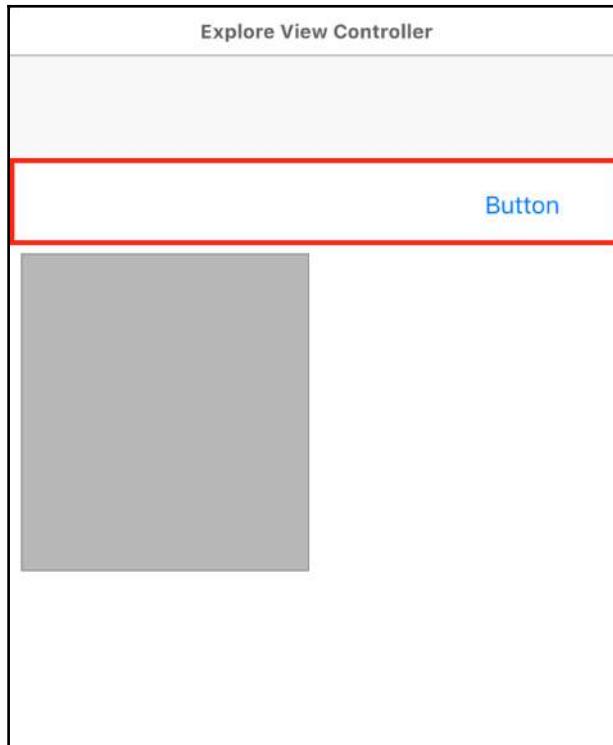
Setting up the Explore header

Let's review the section header for the **Explore** tab:



In this header, we only have four elements: two `UILabels` (title and subtitle), a button, and a gray line underneath the title and button.

We already have the button in the prototype header (collection reusable view), which we created in [Chapter 8, Building Our App Structure in Storyboard](#), and now we need to add the two `UILabels` and then revise all three elements so that they match our design:

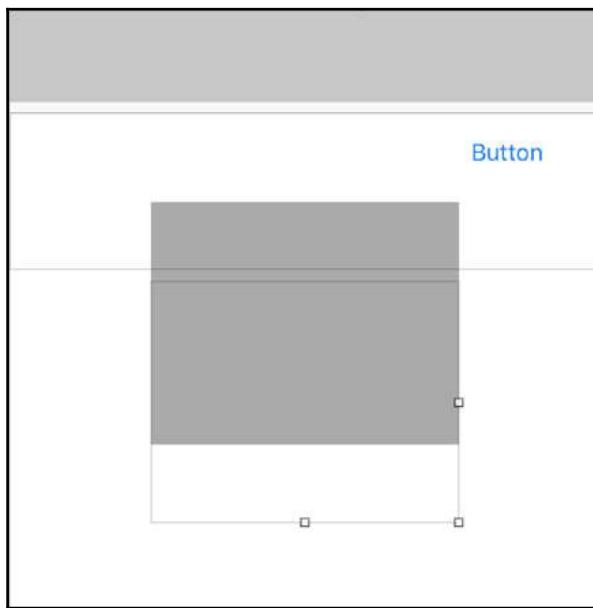


When working with multiple components in the same area, I like to put them into a view. The view acts as a container and allows me to keep my constraints down. Let's get started:

1. In `Main.storyboard`, select the prototype header and, in the Size inspector, update the following values:

- **Width:** 0
- **Height:** 100

When you update the size, you might experience the following:



If you do, click on a different file and come back; the storyboard should fix itself.

2. In the filter field of the object library, type `view`.

3. Drag out a **View** into the prototype header.



Make sure you use the outline view and move this below all the other elements using the Outline view. If you do not, it will cover everything.

4. In the Size inspector, update the following values:

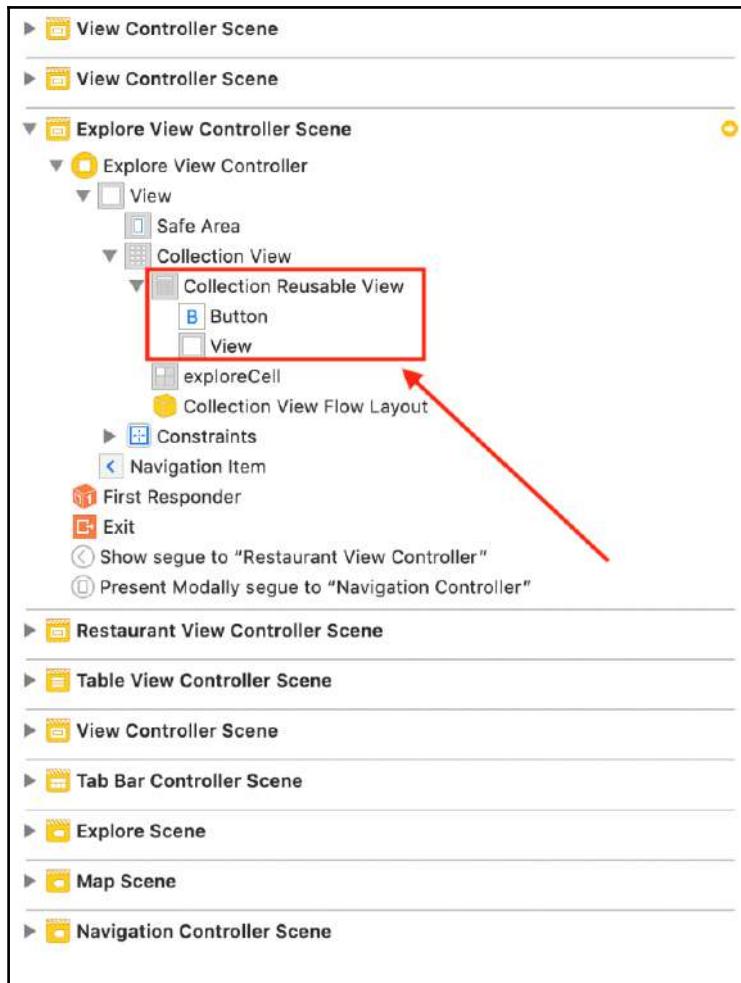
- **X:** 0
- **Y:** 0
- **Width:** 375
- **Height:** 90

At this point, this view is covering up our button.

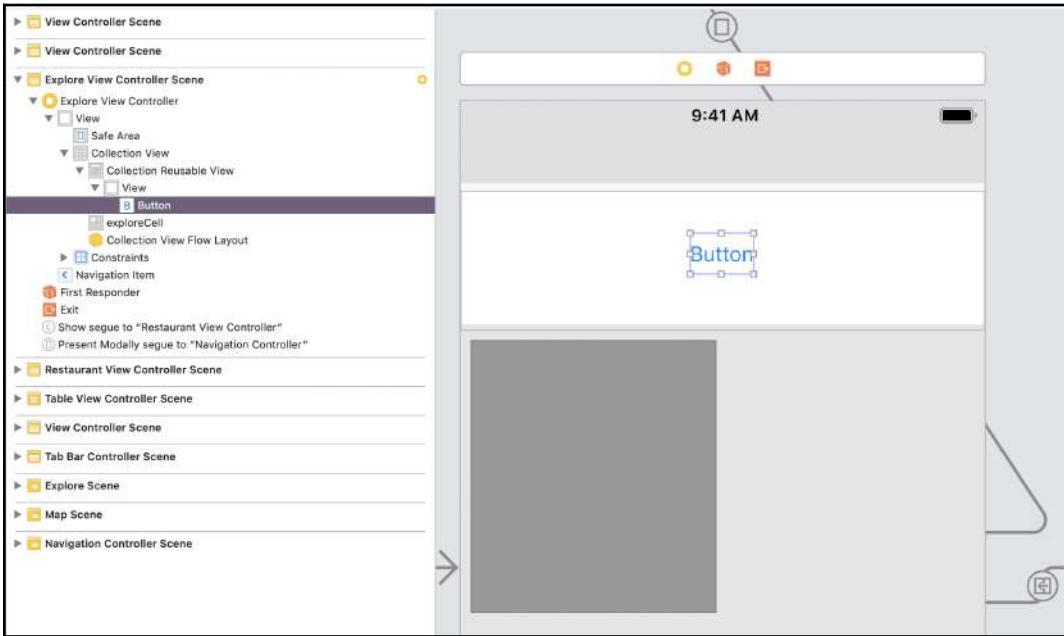


Let's fix this next.

5. Expand the Outline and you will see our newly added View and Button:



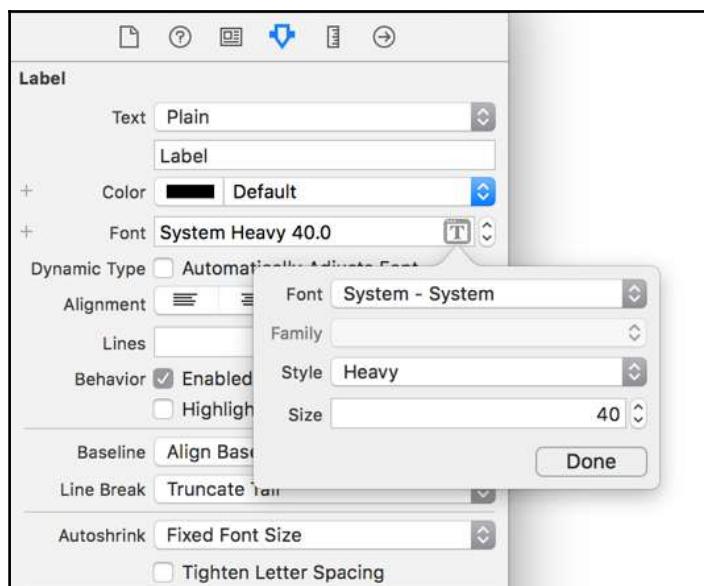
6. Select the **Button** in the Outline and drag it into the **View**. When you are done, it should look like the following:



7. Next, type `label` in the filter field of the object library.
8. Drag out two **Labels** into the **View** of the prototype header that we just added. When you are done, you should see the following:



9. We are going to add a new color to our `Assets.xcassets` file. Name the color LetsEat Light Grey and set the **Hex Color #** to AFAFB2.
10. Let's rename Demo Grey to LetsEat Dark Grey. Our cells do not update to this new color, yet it does not break anything. We will change these colors later in the chapter.
11. Now, select one of the labels, which will be our subtitle, and, in the Attributes inspector, update the following values:
 - **Color:** LetsEat Light Grey
 - **Font:** System Semibold 13
12. In the Size inspector, update the following values:
 - **X:** 8
 - **Y:** 24
 - **Width:** 350
 - **Height:** 21
13. Select the other label, which will be our title, and, in the Attributes inspector, update the value of **Font** to System Heavy 40.0:

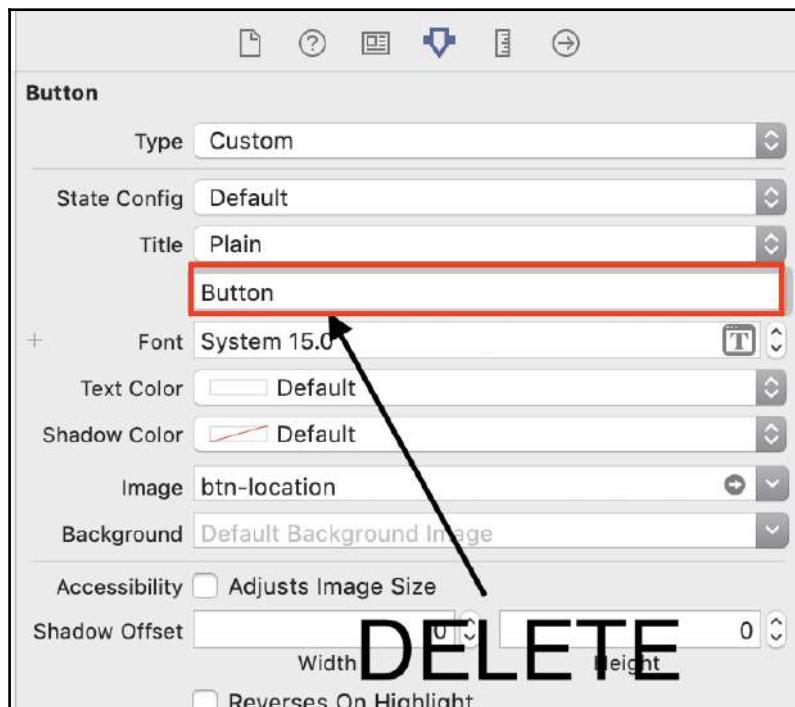


14. In the Size inspector, update the following values:

- **X:** 8
- **Y:** 45
- **Width:** 255
- **Height:** 37

15. Select the button (you might have to select it from the outline if the labels are covering it); then, in the Attributes inspector, update the following values:

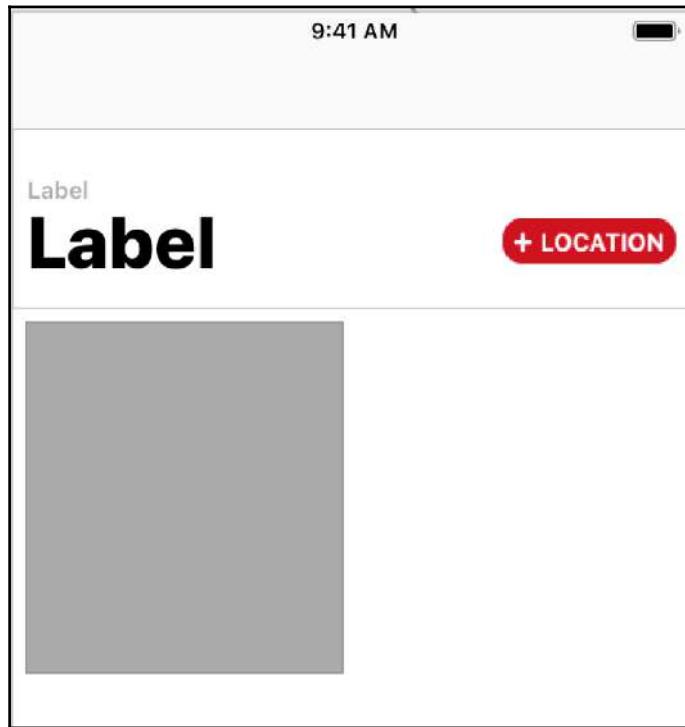
- **Type:** Custom
- **Image:** btn-location
- Remove the text button



16. In the Size inspector, update the button to the following values:

- **X:** 271
- **Y:** 50

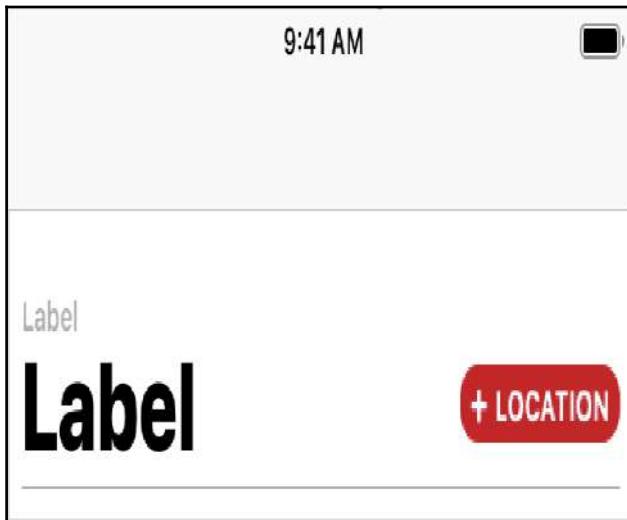
You should now have the following:



17. Type `view` in the filter field of the object library.
18. Drag a View into our view that we are using as a container.
19. Select the View (make sure you use the outline view and move this below all the other elements) and, in the Size inspector, update the following values:
 - **X:** 8
 - **Y:** 89
 - **Width:** 359
 - **Height:** 1

20. Now, with the View selected, in the Attributes inspector, update the value of **Background** to LetsEat Light Grey:

Now, with all of the elements placed into the prototype header, your cell should look as follows (inside your label, you should have the text **Label**):



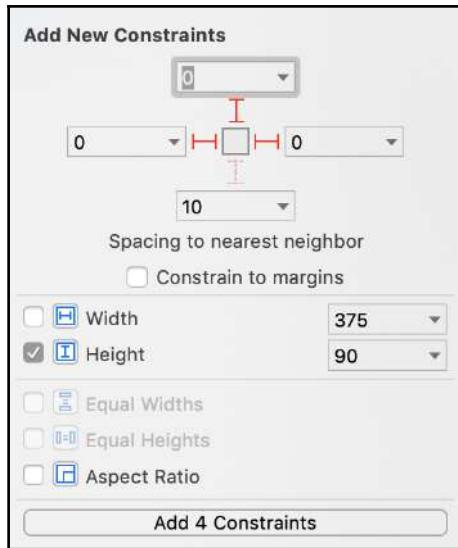
To ensure that our cells adjust to the size of different devices, we must add Auto Layout constraints.

Adding Auto Layout to the Explore header

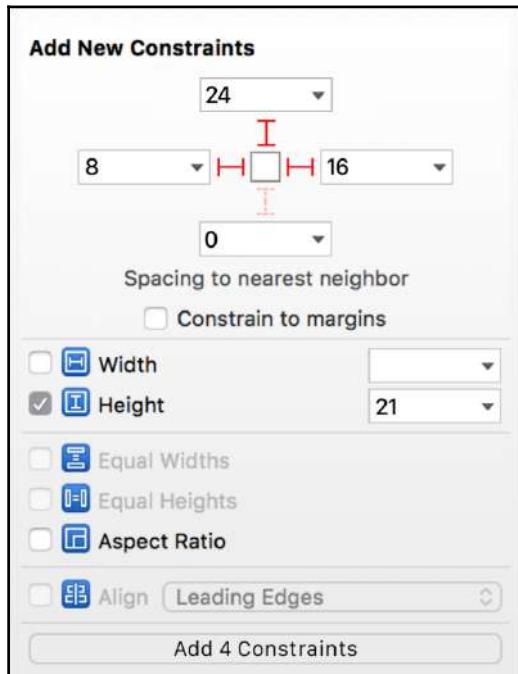
Working with Auto Layout can be very frustrating. If it does not work correctly, I recommend that you clear all the constraints and start over.

Let's begin by adding Auto Layout to our Label subtitle, which is where we should show the currently selected location:

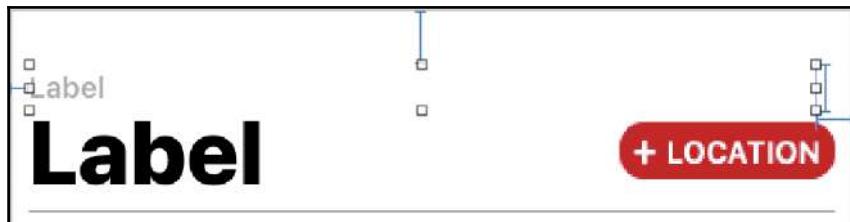
1. Select the View, that we are using as a container and then the Pin icon. Enter the following value:



2. Click **Add 4 Constraints**.
3. Select the Label subtitle and then the Pin icon. Enter the following values:

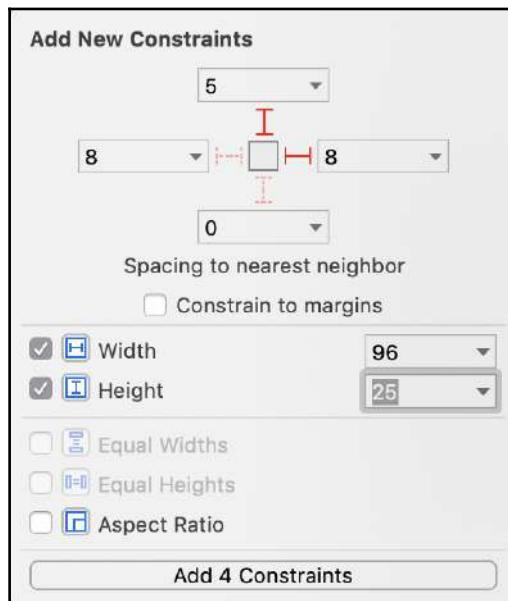


4. Click **Add 4 Constraints**. Your constraints should look like the following:



5. Next, select the **Location** button and then the Pin icon. Enter the following values:

- **Top:** 5
- **Right:** 8
- **Constrain to margins:** Unchecked
- **Width:** 96 (should be checked)
- **Height:** 25 (should be checked)



6. Click **Add 4 Constraints**. You should now see the following:



7. Now, select the grey line (it might be easier to use the outline to select the view) and then the Pin icon. Enter the following values:

- **Right:** 8
- **Bottom:** 0
- **Left:** 8
- **Constrain to margins:** Unchecked
- **Height:** 1 (should be checked)

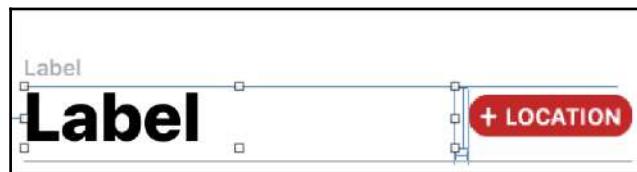
8. Click **Add 4 Constraints**. Your constraints should look like the following:



9. Select the **Label** (title) and then the Pin icon. Enter the following values:

- **Top:** 0
- **Right:** 8
- **Left:** 8
- **Height:** 37 (should be checked)
- **Constrain to margins:** Unchecked

10. Click **Add 4 Constraints**. You should see the following when you are done:



We have completed adding Auto Layout to the **Explore** tab header. If you want to check every constraint, feel free to look at [Chapter 11, Getting Started with the Grid](#), starter project. Let's look at designing the Explore Cell next.

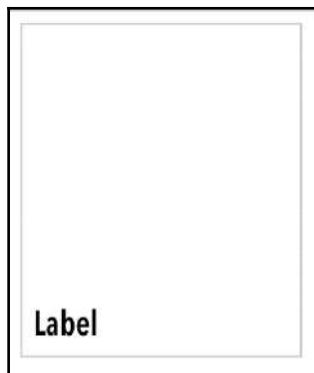
Setting up the Explore cell

Next, let's work on the Explore Collection View cell:

1. Select the prototype cell, called `exploreCell`, in the Attributes inspector, and update the background color to white. Then, in the Size inspector, change the Size from Default to Custom.
2. Then update the following values:
 - **Width:** 176
 - **Height:** 195
3. In the object library's filter field, type `view`.
4. Drag a View into the prototype cell.
5. Select the View and, in the Size inspector, update the following values:
 - **X:** 0
 - **Y:** 0
 - **Width:** 176
 - **Height:** 156
6. Type `image` in the filter field.
7. Drag an Image view into the View we just added.
8. With the Image view still selected, update the following values in the Size inspector:
 - **X:** 0
 - **Y:** 0
 - **Width:** 176
 - **Height:** 156
9. Type `label` in the filter field.
10. Drag a Label into the prototype cell (not the View).

11. With the Label selected, update the value of **Font** in the Attributes inspector to **Avenir Next Condensed Demibold 20**.
12. In the Size inspector, update the following values:
 - **X:** 8
 - **Y:** 165
 - **Width:** 160
 - **Height:** 21

`exploreCell` is now complete. Your cell should now look like the following:



When setting up elements in storyboard, I like to get all my sizes set and then I use Auto Layout constraints to make sure it works for all devices. Let's add Auto Layout constraints before we move on to our Restaurant cell.

Adding Auto Layout to the Explore cell

1. In the Outline view, select the container View that is holding the Image view and then the Pin icon. Enter the following values:
 - **Top:** 0
 - **Right:** 0
 - **Left:** 0
 - **Constrain to margins:** Unchecked
 - **Height:** 156 (should be checked)

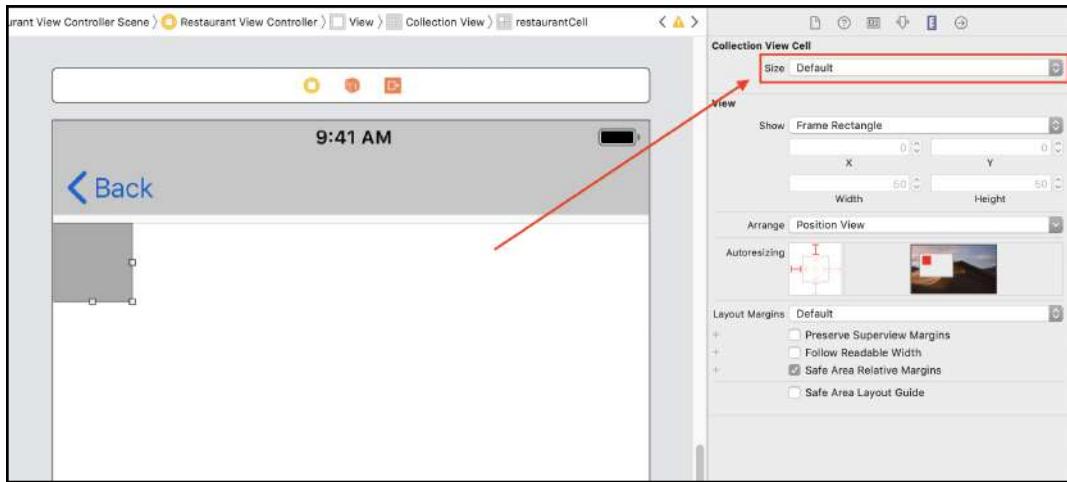
2. Click **Add 4 Constraints**.
3. Select the Image view and then the Pin icon. Enter the following values:
 - **Top:** 0
 - **Right:** 0
 - **Bottom:** 0
 - **Left:** 0
 - **Constrain to margins:** Unchecked
4. Click **Add 4 Constraints**.
5. Select the Label in this `exploreCell` and then the Pin icon. Enter the following values:
 - **Top:** 9
 - **Right:** 8
 - **Left:** 8
 - **Constrain to margins:** Unchecked
 - **Height:** 21 (should be checked)
6. Click **Add 4 Constraints**.

The Explore cell now has all the necessary constraints, and we can now set up the Restaurant cell.

Setting up the Restaurant cell

The Restaurant cell that we are setting up has many elements, so make sure to take your time. Make sure that you go to the Restaurant View Controller; let's get started:

1. Select the prototype cell, called `restaurantCell`, in the Attributes inspector, and update the background color to white. Then, in the Size inspector, change the **Size** from **Default** to **Custom**:

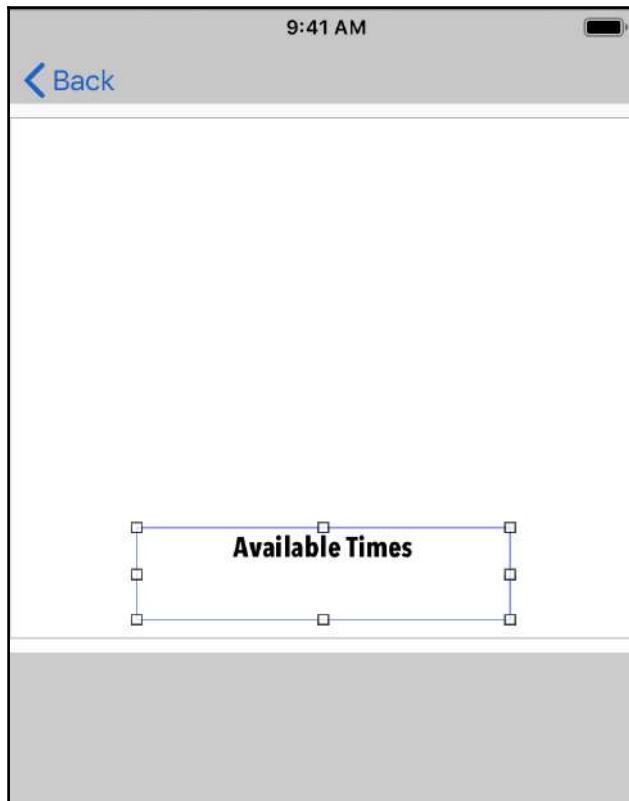


2. Then, update the following values:
 - **Width:** 375
 - **Height:** 312
3. In the filter field of the object library, type `view`.
4. Drag a View into the prototype cell.
5. With the View selected, update the following values in the Size inspector:
 - **X:** 75.5
 - **Y:** 245
 - **Width:** 224
 - **Height:** 56
6. Type `label` in the filter field.
7. Drag a **Label** into the **View** we just added.
8. With the Label selected, update the following values in the Size inspector:
 - **X:** 0
 - **Y:** 2
 - **Width:** 224
 - **Height:** 21

9. In the Attributes inspector, update the following values:

- **Text:** Add **Available Times** into the empty text field under the **Text**
- **Color:** Black Color
- **Alignment:** Center
- **Font:** Avenir Next Condensed Bold 17

When you are done, you should see the following:



10. Next, in the filter field of the object library, type **button**.
11. From the Object library, drag the button from the into the View where we have the label.

12. With the button selected, update the following values in the Attributes inspector:

- **Type:** System
- **Title:** Plain and then add **7:30 pm** in the empty text field under the **Title**
- **Font:** Avenir Next Condensed Regular 17
- **Text Color:** White Color
- **Background:** time-bg

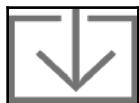
13. In the Size inspector, update the following values:

- **Width:** 68
- **Height:** 27

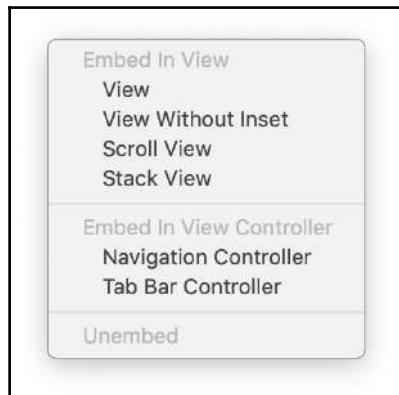
14. Select the button in the Outline view and hit *command + C* to copy.

15. Hit *command + V* twice to paste. You should now have three buttons.

16. Using the Outline view, *command + click* each button created and click on the **Embed in View** icon. The stack icon is two icons to the left of the Pin icon:



17. Select **Stack View** in the dropdown:

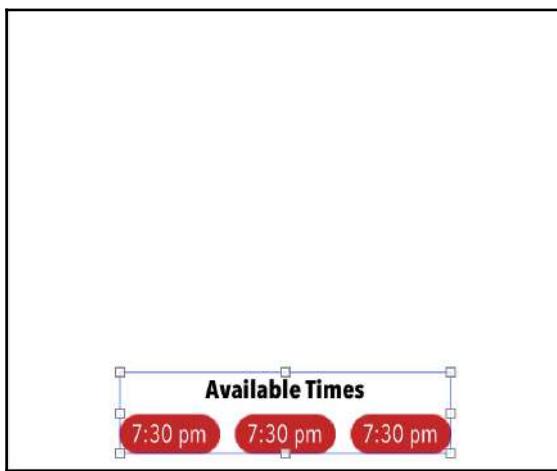


18. Select the stack view in the Outline view, and update the following values in the Attributes inspector:

- **Axis: Horizontal**
- **Alignment: Fill**
- **Distribution: Equal Spacing**
- **Spacing: 10**

19. In the Size inspector, set **X** to 0 and **Y** to 29.

When you are done, you should see the following:



20. In the filter field of the Object library, type `view`.

21. Drag a View into the prototype cell.

22. With the View selected, update the following values in the Size inspector:

- **X: 11**
- **Y: 42**
- **Width: 353**
- **Height: 200**

23. Type `image` in the filter field.

24. Drag out an Image view into the View we just added.

25. Select the Image view in the Outline view and update the value of **Image** in the Attributes inspector with `american`. We are just using this image as a placeholder to see that our cells are set up correctly. Later, we will remove this and load the images using code.
26. With the Image view selected, update the following values in the Size inspector:
 - **X:** 0
 - **Y:** 0
 - **Width:** 353
 - **Height:** 200
27. Type `label` in the filter field.
28. Drag two Labels into the prototype cell.
29. Select one of the Labels and update the value of **Font** in the Attributes inspector with **Avenir Next Condensed Demi Bold 17**.
30. In the Size inspector, update the following values:
 - **X:** 10
 - **Y:** 3
 - **Width:** 355
 - **Height:** 19
31. Select the other Label and update the following values in the Attributes inspector:
 - **Color:** LetsEat Dark Grey
 - **Font:** Avenir Next Condensed Regular 14
32. In the Size inspector, update the following values:
 - **X:** 10
 - **Y:** 22
 - **Width:** 355
 - **Height:** 16

We have completed our Restaurant cell setup, and it now looks like the following:



Now we need to add Auto Layout to all of the elements.

Adding Auto Layout to the Restaurant cell

Since we have many elements in the Restaurant cell, it means that there are more chances for errors with Auto Layout. Although you may get frustrated when using Auto Layout, if you are a visual person like me, hopefully, you will eventually appreciate using it:

1. Select the top label and then enter the following values:

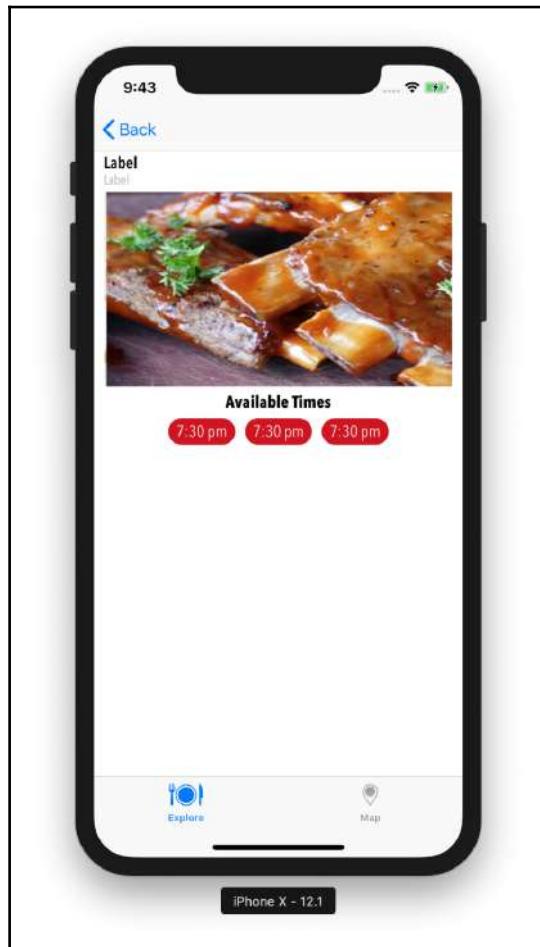
- **Top:** 3
- **Right:** 10
- **Left:** 10
- **Constrain to margins:** Unchecked
- **Height:** 19 (should be checked)

2. Click **Add 4 Constraints**.
3. Select the Label right under the last label and then the Pin icon. Enter the following values:
 - **Top:** 0
 - **Right:** 10
 - **Left:** 10
 - **Constrain to margins:** Unchecked
 - **Height:** 16 (should be checked)
4. Click **Add 4 Constraints**.
5. Select the Image container and then the Pin icon. Enter the following values:
 - **Top:** 4
 - **Constrain to margins:** Unchecked
 - **Width:** 353 (should be checked)
 - **Height:** 200 (should be checked)
6. Click **Add 3 Constraints**.
7. Click on the Align icon and enter the value of **Horizontally in Container** as 0 (this should be checked).
8. Click **Add 1 Constraint**.
9. Select the Image inside of the container and then the Pin icon. Enter the following values:
 - **Top:** 0
 - **Right:** 0
 - **Bottom:** 0
 - **Left:** 0
 - **Constrain to margins:** Unchecked
10. Click **Add 4 Constraints**.

11. Select the container that is holding the stack view and the available time's label and then select the Pin icon. Enter the following values:
 - **Top:** 3
 - **Constrain to Margins:** Unchecked
 - **Width:** 224
 - **Height:** 56
12. Click **Add 3 Constraints**.
13. Click on the Align icon and enter the **Horizontally in Container** value as 0 (this should be checked).
14. Click **Add 1 Constraint**.
15. Select the stack view inside of the container and then the Pin icon. Enter the following values:
 - **Top:** 6
 - **Right:** 0
 - **Left:** 0
 - **Constrain to margins:** Unchecked
 - **Height:** 27
16. Click **Add 4 Constraints**.
17. Select the Label that is above the three buttons and then the Pin icon. Enter the following values:
 - **Top:** 2
 - **Left:** 0
 - **Right:** 0
 - **Constrain to margins:** Unchecked
 - **Height:** 21
18. Click **Add 4 Constraints**.
19. Click on the Align icon and enter the value of **Horizontally in Container** as 0 (this should be checked).
20. Click **Add 1 Constraint**.

Now, all of the Auto Layout for the Restaurant cell is set up. Before you run the project, select the collection view in the outline. Then, go to the Size inspector and update Cell Size **Width** to 375 and **Height** to 312. These numbers are used for design purposes, and we will make these values dynamic depending on the device size later.

Let's build and run our project and go to the restaurant cell. You should now see the following:

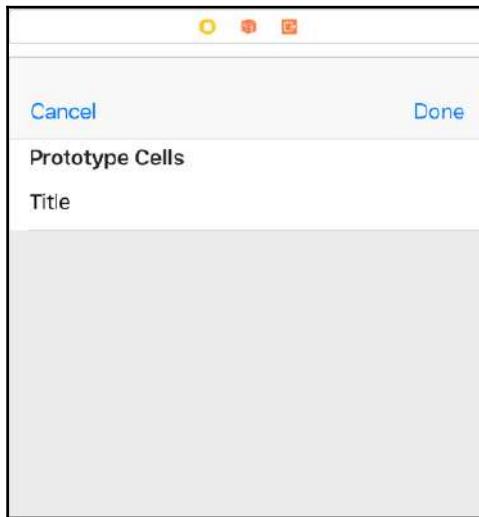


The Locations cell

We now need to work on the Locations cell. Find the view we will use for our locations and, in your table view we need to update this cell, for this cell, we are using a predefined cell that Apple provides. Let's update our Table View by do the following:

1. Select the Table View, and update **Prototype Cells** to 1.
2. Select the prototype cell and enter the following values:
 - **Style: Basic**
 - **Identifier:** locationCell

That is all we need to do. Now, your cell should look as follows:



When you change the style from **Custom** to **Basic**, the word **Title** should appear in the cell. The word **Title** is just placeholder text. We have now finished designing our cell.

Summary

In this chapter, we formatted our cells to match our design and added Auto Layout constraints. Auto Layout can be complicated; however, as with anything, the more you practice, the easier it gets. You can write Auto Layout with code, but it is not what I prefer in a storyboard. If you would like to do it in code, there are plenty of tutorials that can help you with this.

We are now finished with the storyboard and design setup, so we can focus on the code side since our UI is pretty much set up. You should have a good idea of how our app should work. If you are struggling, there is nothing wrong with that. This stuff takes time to click, and, as I have said before, if you are struggling with anything that we have done, please go back. If you keep going when you are not comfortable, learning will get harder and harder. We will be covering a lot of new topics, so adding on to them when you are not ready is not recommended.

In the next chapter, we will learn what the Model View Controller is and how to work with it.

11

Getting Started with the Grid

I am a visual person; I prefer to start with the visuals and make sure that the app looks like the design. Starting with the UI helps me to identify the data structure and allows me to get familiar with the app, which means I can then focus my attention on the code.

In earlier chapters, we set up our app structure and developed a good understanding of the basics involved. In this chapter, you will learn about app architecture and how to create it for our *Let's Eat* app. For this chapter, please use the Chapter 11 project files, as I have added more design elements that you will need throughout the book.

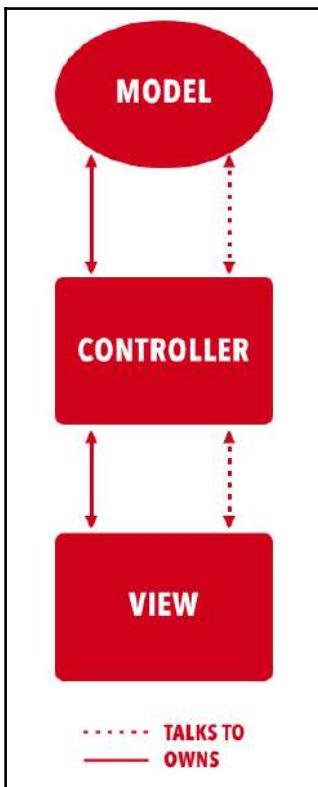
We will cover the following in this chapter:

- Understanding the Model View Controller architecture
- Classes and structures
- Controllers and classes

Understanding the Model View Controller architecture

Apple built iOS apps to use what is known as the **Model View Controller (MVC)**, which is an architectural pattern that describes a way to structure the code in your app. In layman's terms, this just means breaking up our app into three distinct camps: Model, View, and Controller.

Here is a diagram of MVC to help you understand it:



Let's discuss each camp:

- **Model:** The **Model** camp is responsible for an app's data and business logic. The Model's only job is to handle representations of data, data storage, and the operations performed on data.
- **View:** The **View** camp is responsible for all the things that you see on the screen. The View handles presenting and formatting data that results from the user's interactions.
- **Controller:** The **Controller** camp is the liaison or coordinator between the other two camps. The Controller handles a lot of setup and connections to the View. The Controller also interprets user interactions. Since the Controller is between both the View and the Model, the View and the Model should know nothing about each other.

In summary, the Controller takes user interactions and either responds back to the View or passes it onto the Model. When the Model completes a task, it passes it back to the Controller, and then the Controller talks with the View.

Getting familiar with the setup

For beginners, the MVC architecture can make you uncertain about where things should go. As we progress through the book, you will learn where to put things and why. So, you need not worry about where things should be placed as we work through this process together, step by step.

As your project grows, the MVC architecture places a lot of the responsibility on the Controller. Therefore, in this book, we tweak the MVC pattern to not put so much pressure on the Controller.

Before we continue with our coding, we need to discuss classes and structures.

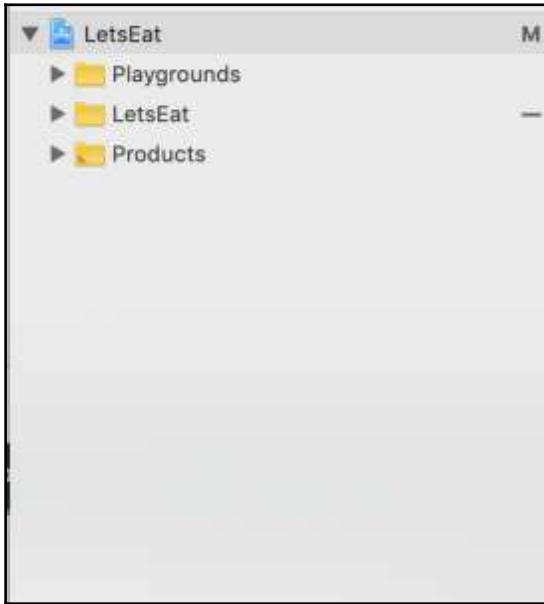
Classes and structures

Classes and structures (also known as structs) are files that contain properties and methods. You use these properties and methods to add functionality. You have been working with structs since [Chapter 1, Getting Familiar with Xcode](#). Strings, Ints, Booleans, Arrays, Dictionaries, and Sets are all structs.

Earlier in the book, we created functions. As noted in [Chapter 6, Starting the UI Setup](#), a method is a function that lives inside a class or struct.

Classes and structs are very similar; however, Swift handles each of them a bit differently. To get a better understanding of how classes and structs work, we create a new Playground project. Working in the Playground gives us the ability to learn how to create custom classes and structs and to gain an understanding of each of their positives and negatives.

Since we already have a project created, we can actually add a playground directly into our project. Right-click in the Project Navigator and create a new group called Playgrounds. When you are done, you should see the following:



Next, right-click on the Playgrounds folder, go to **New File**, and do the following:

1. Scroll to the bottom of the template screen, select a **Blank** playground, and hit **Next**.
2. In the options screen that appears, name your new **Playground FunctionsStructs**, and make sure that your Platform is set to iOS. Hit **Next** and then **Create**. Now, let's delete everything inside your new Playground and toggle on the **Debug** Panel, using either the toggle button or *command + shift + Y*.

In your empty Playground, add the following:

```
class Cat {  
}  
  
struct Dog {  
}
```

We just created our first class and struct and defined two new custom data types (known as **Swift types**), `Cat` and `Dog`. Since we have not yet given the class or struct a property (such as a name) or created an instance of either `Cat` or `Dog`, you see nothing in the **Results** or **Debug** Panels.

When you create classes and structs, it is best practice to start with a capital letter. Also, you must have different names for your class and your struct. Otherwise, you will get an error. Even though one is a class and the other is a struct, each of them needs a distinct name.

Now, we need to give names to our `Cat` class and our `Dog` struct. Therefore, let's give them both a property, called `name`:

```
class Cat {  
    var name:String?  
}  
  
struct Dog {  
    var name:String?  
}
```

If you cannot set a property when it is created, then it is recommended that you set that property to an optional using the question mark (?). Using optional protects your code from trying to access the name if you never set it. You can also set your variable as an optional unwrapped. For example, you can also do the following:

```
var name:String!
```

With both `Cat` and `Dog` now having a property called `name`, let's create an instance of each of them:

```
let yellowCat = Cat()
yellowCat.name = "Whiskers"
print(yellowCat.name as Any)

var yellowDog = Dog()
yellowDog.name = "Bruno"
print(yellowDog.name as Any)
```

The screenshot shows a Xcode Playgrounds interface. On the left, there is a code editor containing Swift code. On the right, there is a preview area showing the resulting objects and their names.

Code Editor Content:

```
class Cat {
    var name:String?
}

struct Dog {
    var name:String?
}

let yellowCat = Cat()
yellowCat.name = "Whiskers"
print(yellowCat.name as Any)

var yellowDog = Dog()
yellowDog.name = "Bruno"
print(yellowDog.name as Any)
```

Preview Area Content:

Object Type	Value
Cat	Cat
Cat	"Optional("Whiskers")\n"
Dog	Dog
Dog	"Optional("Bruno")\n"

Output Console:

```
Optional("Whiskers")
Optional("Bruno")
```

So far, everything on the surface looks the same. We created both a `Cat` and a `Dog` and gave them each names. However, let's say `Whiskers` runs away and, a few weeks later, finds a home with a new family, who decide to change his name to `Smokey`. After `Whiskers` runs away, `Bruno` becomes lonely and decides to find him, but also gets lost. `Bruno` finds a new home as well, and this new family decides to name him `Max`.

In Playgrounds, we create a new constant called `yellowStrayCat` and set it equal to `yellowCat`, since it is still `Whiskers`. However, we change the name of `yellowStrayCat` to `Smokey`. We also create a new constant called `yellowStrayDog`, setting it equal to `yellowDog` and naming it `Max`:

```
let yellowStrayCat = yellowCat
yellowStrayCat.name = "Smokey"
print(yellowStrayCat.name)

var yellowStrayDog = yellowDog
yellowStrayDog.name = "Max"
print(yellowStrayDog.name)
```

Our **Results** Panel shows that the names of `yellowStrayCat` and `yellowStrayDog`, respectively, are now `Smokey` and `Max`. So, everything seems to be the same between our class and our struct, right? No, they are not the same. Let's print the name of `yellowCat` underneath the line where we have `print(yellowStrayCat.name)`. In addition, let's do the same for the name of `yellowDog` underneath where we have `print(yellowStrayDog.name)`. Your code should now look as follows:

```
let yellowStrayCat = yellowCat
yellowStrayCat.name = "Smokey"
print(yellowStrayCat.name)
print(yellowCat.name)

var yellowStrayDog = yellowDog
yellowStrayDog.name = "Max"
print(yellowStrayDog.name)
print(yellowDog.name)
```

In our **Results** Panel, you should notice an unexpected result. The `yellowCat`, `Whiskers`, now has the name `Smokey`, but the `yellowDog` is still `Bruno`. Without getting too technical, when you use a class and copy it as we did, it refers back to the original instance created. This is known as a **reference type**. However, when structs get copied, they create a new instance and the original is not affected. This is known as a **value type**.

Before we move on, let's look at one more difference between the two. In programming, we have what is called **inheritance**, which means that we can create another object with default values and other objects can inherit from those default values. Let's create an `Animal` class that is the base class immediately below our `Cat` class:

```
class Animal {
    var age:Int?
}
```

Now, let's update our `Cat` class to inherit from it, as shown in the following code:

```
class Cat:Animal {  
    ...  
}
```

Note that we are only updating what goes directly after `Cat`. The rest of the class in the curly brackets stays the same.

Since our class now inherits from `Animal`, we should have a new property called `age`. Underneath where we name `yellowCat` as `Whiskers` and above our `print` statement, enter the following after we set `Whiskers'` name:

```
yellowCat.age = 3
```

The screenshot shows a Swift playground interface. On the left, there is a code editor containing the following Swift code:

```
class Animal {  
    var age:Int?  
}  
  
class Cat:Animal {  
    var name:String?  
}  
  
struct Dog {  
    var name:String?  
}  
  
let yellowCat = Cat()  
yellowCat.name = "Whiskers"  
yellowCat.age = 3  
print(yellowCat.name as Any)  
  
var yellowDog = Dog()  
yellowDog.name = "Bruno"  
print(yellowDog.name as Any)  
  
let yellowStrayCat = yellowCat  
yellowStrayCat.name = "Smokey"  
print(yellowStrayCat.name as Any)  
print(yellowCat.name as Any)  
  
var yellowStrayDog = yellowDog  
yellowStrayDog.name = "Max"  
print(yellowStrayDog.name as Any)  
print(yellowDog.name as Any)
```

A red arrow points from the line `yellowCat.age = 3` to the right pane, which displays the results of the playground's execution. The right pane has two columns of output for each variable assignment:

Variable	Value
<code>Cat</code>	<code>Cat</code>
<code>Cat</code>	<code>"Optional("Whiskers")\n"</code>
<code>Dog</code>	<code>Dog</code>
<code>Dog</code>	<code>"Optional("Bruno")\n"</code>
<code>Cat</code>	<code>Cat</code>
<code>Cat</code>	<code>"Optional("Smokey")\n"</code>
<code>Cat</code>	<code>"Optional("Smokey")\n"</code>
<code>Dog</code>	<code>Dog</code>
<code>Dog</code>	<code>"Optional("Max")\n"</code>
<code>Dog</code>	<code>"Optional("Bruno")\n"</code>

Below the main output area, there is a small sidebar with several icons representing different types of playground content.

So, as expected, we were able to give `Whiskers` an age. Let's do the same for our `Dog` struct by adding `Animal` directly after `Dog`:

```
struct Dog:Animal {  
    var name:String?  
}
```

Once you have entered the preceding code snippet, you will see the following:

The screenshot shows a Swift playground interface. On the left, there is a code editor containing the following code:

```
class Animal {
    var age:Int?
}

class Cat:Animal {
    var name:String?
}

struct Dog:Animal { // Non-class type 'Dog' cannot inherit from class 'Animal'
    var name:String?
}

let yellowCat = Cat()
yellowCat.name = "Whiskers"
yellowCat.age = 3
print(yellowCat.name as Any)

var yellowDog = Dog()
yellowDog.name = "Bruno"
print(yellowDog.name as Any)

let yellowStrayCat = yellowCat
yellowStrayCat.name = "Smokey"
print(yellowStrayCat.name as Any)
print(yellowCat.name as Any)

var yellowStrayDog = yellowDog
Optional("Whiskers")
Optional("Bruno")
Optional("Smokey")
Optional("Smokey")
Optional("Max")
Optional("Bruno")
```

A red error message is displayed in the code editor: **Non-class type 'Dog' cannot inherit from class 'Animal'**. To the right of the code editor, there is a preview pane showing the state of variables:

Variable	Type	Value
yellowCat	Cat	Cat
yellowCat.name	"Optional("Whiskers")"	"Optional("Whiskers")"\n
yellowDog	Dog	Dog
yellowDog.name	"Optional("Bruno")"	"Optional("Bruno")"\n
yellowStrayCat	Cat	Cat
yellowStrayCat.name	"Optional("Smokey")"	"Optional("Smokey")"\n"Optional("Smokey")"\n
yellowStrayDog	Dog	Dog

A red error displays and informs you that Non-class type 'Dog' cannot inherit from class 'Animal'. Therefore, we need to create a struct called `AnimalB`, since structs cannot have the same name:

```
struct AnimalB {
    var age:Int?
}
```

Update your `Dog` struct from `Animal` to `AnimalB`:

```
struct Dog:AnimalB {
    var name:String?
}
```

Now, you should see an error, Inheritance from non-protocol type 'AnimalB', which means that our struct cannot inherit from another struct:

The screenshot shows a Swift playground in Xcode. On the left, there is code defining classes and structs. A specific line of code for a struct named 'Dog' is highlighted, and a red box with the text 'Inheritance from non-protocol type 'AnimalB'' appears above it. To the right of the code, there is a vertical stack of output text boxes. The first three boxes show the creation of 'Cat' and 'Dog' instances, each followed by an optional string value. The fourth box shows the creation of a 'Dog' instance, also followed by an optional string value. The fifth box shows the creation of a 'Cat' instance, followed by an optional string value. The bottom-most box shows the creation of two more 'Dog' instances, each followed by an optional string value.

```
class Animal {
    var age:Int?
}

class Cat:Animal {
    var name:String?
}

struct AnimalB {
    var age:Int?
}

struct Dog:AnimalB {           // Inheritance from non-protocol type 'AnimalB'
    var name:String?
}

let yellowCat = Cat()
yellowCat.name = "Whiskers"
yellowCat.age = 3
print(yellowCat.name as Any)

var yellowDog = Dog()
yellowDog.name = "Bruno"
print(yellowDog.name as Any)

let yellowStrayCat = yellowCat
yellowStrayCat.name = "Smokey"
print(yellowStrayCat.name as Any)
print(yellowCat.name as Any)

Optional("Whiskers")
Optional("Bruno")
Optional("Smokey")
Optional("Smokey")
Optional("Max")
Optional("Bruno")
```

Cat	Cat	Cat	"Optional("Whiskers")\n"
Dog	Dog	Dog	"Optional("Bruno")\n"
Cat	Cat	Cat	"Optional("Smokey")\n"
"Optional("Smokey")\n"			
Optional("Whiskers")	Optional("Bruno")	Optional("Smokey")	
Optional("Smokey")	Optional("Max")	Optional("Bruno")	

Inheritance is something that you can do with classes, but not with structs; this is another difference between classes and structs. There are a couple of other advanced technical differences but, for our purposes, the two described here are sufficient.

Controllers and classes

When working with `UIViewController`, `UICollectionViewController`, and `UITableViewController`, you need to create a class file for each of these elements. Each file handles all of the logic and interactions that the controller sends and receives. Along with interactions, the class file is responsible for receiving data. We can see what this looks like in Playgroud. Let's see how this works:

1. Right-click on the Playgrounds folder and go to **New File**.

2. Scroll to the bottom of the template screen, select a **Blank** playground, and hit **Next**.
3. In the options screen that appears, name your new Playground `CollectionViewBasics`, and make sure that your Platform is set to iOS. Hit **Next** and then **Create**. Delete the variable and leave the import statement, then toggle open/on the **Debug** Panel, using either the toggle button or *command + shift + Y*.

Now that we are set up, let's see how we can view the UI inside of playgrounds. Using Playgrounds really lets us focus on learning, instead of, having to worry about running our project every time we want to see changes.

Understanding Collection Views

The first thing we want to do is get access to all of the UI and Playground components that we will need. At the very top of the playground, please add the following import statements:

```
import UIKit
import PlaygroundSupport
```

The first import statement imports all of the UI elements we will need. The second import gives us access to Playground support, this will allow us to add our UI elements into playgrounds. Now let's create our first `UIViewController`, this setup is pretty much the same structure I like to use for all of my classes that are controllers. Add a line break and then add the following code:

```
class CollectionViewExampleController: UIViewController {  
}
```

This code looks pretty similar to what we discussed earlier in this chapter. We created a class named `CollectionViewExampleController` and we subclass `UIViewController`. `UIViewController` is a class that Apple provides us with, and it gives us access to a lot of things. Going into all of them would take a chapter in itself, but throughout the book, we will slowly introduce you to new things we can access. Next, we need to create `UICollectionView`, so let's do that next by adding the following code inside of our curly brackets:

```
var collectionView: UICollectionView?
```

After you add the variable, your complete code should look like the following:

```
class CollectionViewExampleController: UIViewController {  
    var collectionView: UICollectionView?  
}
```

This is the variable we will use for our UICollectionView. The question mark signifies that we are making it an **Optional** value, just as we discussed earlier. The next thing we should do is the UICollectionViewDataSource protocol. This protocol has many methods that we can use, but there are two that are required to use UICollectionView. This protocol allows us to tell collectionView how many items we have in each section of collectionView, as well as creating a cell for each item we want to display in our collection view. Update your class to conform to UICollectionViewDataSource by adding it after the subclassing of UIViewController. In order to add it after, we must add a comma after UIViewController. When you are done, you should have the following:

```
class CollectionViewExampleController: UIViewController,  
UICollectionViewDataSource {  
    var collectionView: UICollectionView?  
}
```

We now understand what this means, but we have an error:

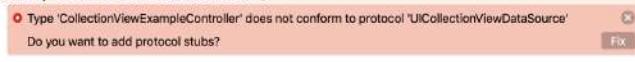
```
1 //: Playground - noun: a place where people can play  
2  
3 import UIKit  
4 import PlaygroundSupport  
5  
6 class CollectionViewExampleController : UIViewController, UICollectionViewDataSource { ⓘ Type 'CollectionViewExampleController' does not conform t...  
7  
8     var collectionView: UICollectionView?  
9  
10 }
```

This error is telling us that we are missing the required methods for UICollectionViewDataSource. Click on the red dot with the white circle in it:

```
1 //: Playground - noun: a place where people can play  
2  
3 import UIKit  
4 import PlaygroundSupport  
5  
6 class CollectionViewExampleController : UIViewController, UICollectionViewDataSource { ⓘ Type 'CollectionViewExampleController' does not conform t...  
7  
8     var collectionView: UICollectionView?  
9  
10 }
```

When we click on the error, it lets us see more details:

```
1 //: Playground - noun: a place where people can play.
2
3 import UIKit
4 import PlaygroundSupport
5
6 class CollectionViewExampleController : UIViewController, UICollectionViewDataSource {
7
8     var collectionView:UICollectionView?
9
10 }
```



We can either add the functions ourselves or we can click on the Fix button and it will add them for us.

Let's do that and we will see that we now have the two required methods needed for UICollectionView. Your code should look like the following:

```
1 //: Playground - noun: a place where people can play
2
3 import UIKit
4 import PlaygroundSupport
5
6 class CollectionViewExampleController : UIViewController, UICollectionViewDataSource {
7
8     func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
9         code
10    }
11
12     func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
13         code
14    }
15
16
17     var collectionView:UICollectionView?
18
19 }
```



As you see, we have the new methods, but now we have more errors.

NOTE: Whenever you use the fix button, it adds the code to the top of your file. When coding, I like to have my variables at the top of my file and my functions after them.

Move the collectionView variable above the newly added functions. Inside each function, you should see the word code:

```
func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
    code ←
}
```

Inside the `numberOfItemsInSection` method, delete the word and replace the code with the following:

```
func collectionView(_collectionView:UICollectionView,
    numberOfRowsInSectionsection:Int) -> Int {
    return 1
}
```

The `return 1` that we added tells our `UICollectionView` that we want to display 1 item.

Next, we need to fix our last method, `cellForItemAt`. Again, this method is responsible for displaying cells in our `UICollectionView`. Let's update this method to display a red box for every item we have in our `collectionView`. Add the following code:

```
func collectionView(_collectionView:UICollectionView,
    cellForItemAtindexPath:IndexPath) -> UICollectionViewCell {
    let cell =
        collectionView.dequeueReusableCell(withIdentifier:"BoxCell", for:indexPath)
    cell.backgroundColor = .red
    return cell
}
```

In the first line of the code we just added, we create a reusable cell with an identifier name of "BoxCell" and we pass the index path of the collection view to the reusable cell. Next, we set the background color to red, and then we return the cell we created to `UICollectionView`. This method is run for every item we need, and in our case, we are returning 1 cell because we set our `numberOfItemsInSection` to 1.

When you are done with both of these methods, your entire file should look like the following:

```
import UIKit
import PlaygroundSupport

class CollectionViewExampleController: UIViewController,
    UICollectionViewDataSource {
    var collectionView:UICollectionView?
    func collectionView(_collectionView:UICollectionView,
        numberOfRowsInSectionsection:Int) -> Int {
        return 1
    }
    func collectionView( _collectionView:UICollectionView,
        cellForItemAtindexPath:IndexPath) -> UICollectionViewCell {
```

```
        let cell =
collectionView.dequeueReusableCell(withIdentifier:"BoxCell",
for:indexPath)
    cell.backgroundColor = .red
    return cell
}
}
```

All of your errors are now gone, but right now, nothing will run because we need to add a couple more things. We have a `collectionView`, but it actually needs to be created before we can use it. Typically, when working, I like to create a method that will create or set up my `collectionView` depending on what I am doing. Let's add the following code after our `collectionView` variable:

```
func createCollectionView() {
    self.collectionView = UICollectionView(frame:CGRect(x:0 , y:0,
width:self.view.frame.width, height:self.view.frame.height),
collectionViewLayout:UICollectionViewFlowLayout())
    self.collectionView?.dataSource = self
    self.collectionView?.backgroundColor = .white
    self.collectionView?.register(UICollectionViewCell.self,
forCellWithReuseIdentifier:"BoxCell")
    self.view.addSubview(self.collectionView!)
}
```

In the first line of this method, we create an instance of `UICollectionView` and we set the frame and layout. The frame sets its x and y positions, as well as, the width and height. Our width and height will match the size of the view we are using. In the next line, we set the data source to `self`, and we already added all of the methods that go with the data source when we set up our `UICollectionViewDataSource`. In the next line, we set the background color of our collection view to white. Then, we register our cell in the next line. In order for the collection view to know about our cell, we have to register it. Finally, we add the collection view to our view. Nothing too complicated here, but it is new and will take a bit of time to get used to. Now we just need to make sure that we call the `createCollectionView` method. Every `UIViewController` has an entry point, and the one that we will use is a method called `viewDidLoad()`. Add the following code above `createCollectionView`:

```
override func viewDidLoad() {
    super.viewDidLoad()
    createCollectionView()
}
```

Here, we are overriding the method `viewDidLoad()` so that we can use it to call methods we need too. Inside this method, we are calling the new method we just created—`createCollectionView()`. We will still not be able to see our `collectionView`, because when you are working inside of your app, this is all the code you would need. Since we are in Playgrounds, we have to do one more thing in order for it to be displayed inside of Playgrounds. After the very last curly bracket, add the following code:

```
// Present the view controller in the Live View window
PlaygroundPage.current.liveView = CollectionViewController()
```

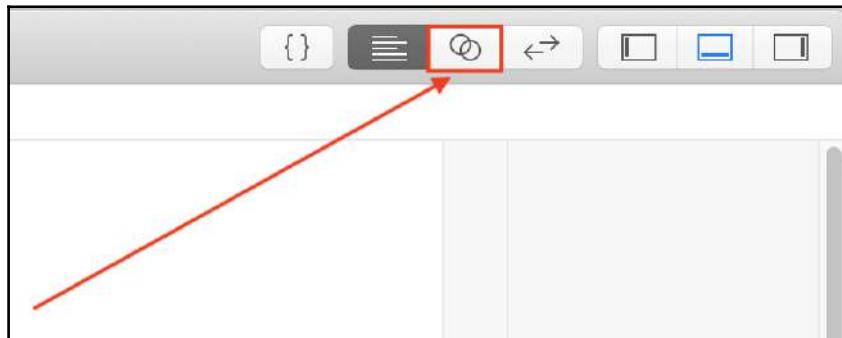
Your entire class should look like the following:

```
import UIKit
import PlaygroundSupport

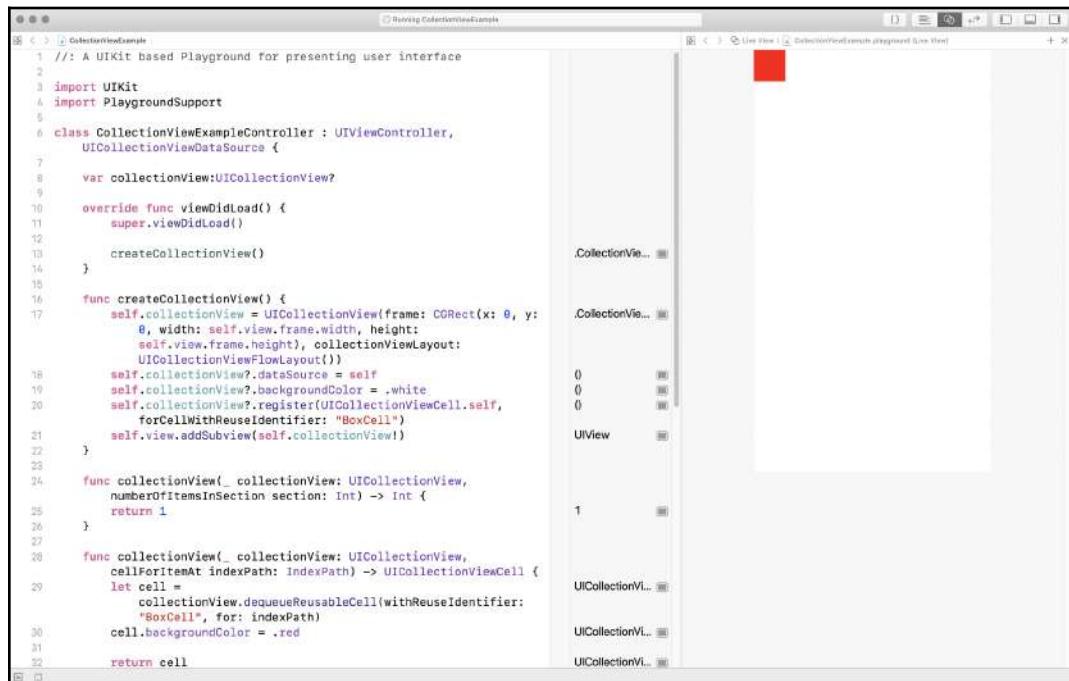
class CollectionViewExampleController: UIViewController,
UICollectionViewDataSource {
    var collectionView: UICollectionView?
    override func viewDidLoad() {
        super.viewDidLoad()
        createCollectionView()
    }
    func createCollectionView() {
        self.collectionView = UICollectionView(frame:CGRect(x:0, y:0,
width:self.view.frame.width,
height:self.view.frame.height),
collectionViewLayout:UICollectionViewFlowLayout())
        self.collectionView?.dataSource = self
        self.collectionView?.backgroundColor = .white
        self.collectionView?.register(UICollectionViewCell.self, forCellWithReu
seIdentifier:"BoxCell")
        self.view.addSubview(self.collectionView!)
    }
    func collectionView(_ collectionView:UICollectionView,
numberOfItemsInSection section:Int) -> Int {
        return 1
    }
    func collectionView(_ collectionView:UICollectionView,
cellForItemAt indexPath:IndexPath) -> UICollectionViewCell {
        let cell =
        collectionView.dequeueReusableCell(withIdentifier:"BoxCell", for:i
ndexPath)
        cell.backgroundColor = .red
        return cell
    }
}
```

```
// Present the view controller in the Live View window  
PlaygroundPage.current.liveView = CollectionViewExampleController()
```

Now, you are probably wondering why you still cannot see anything, and that is because we need to open up the Assistant Editor. Click the following icon to do so:



When you do this, you will see your collection view and 1 red box on the screen:



Note that you might have to click the play button at the bottom of your code to get it to show up:

```
32  }
33 // Present the view controller in the Live View window
34 PlaygroundPage.current.liveView =
    CollectionViewExampleController()
```

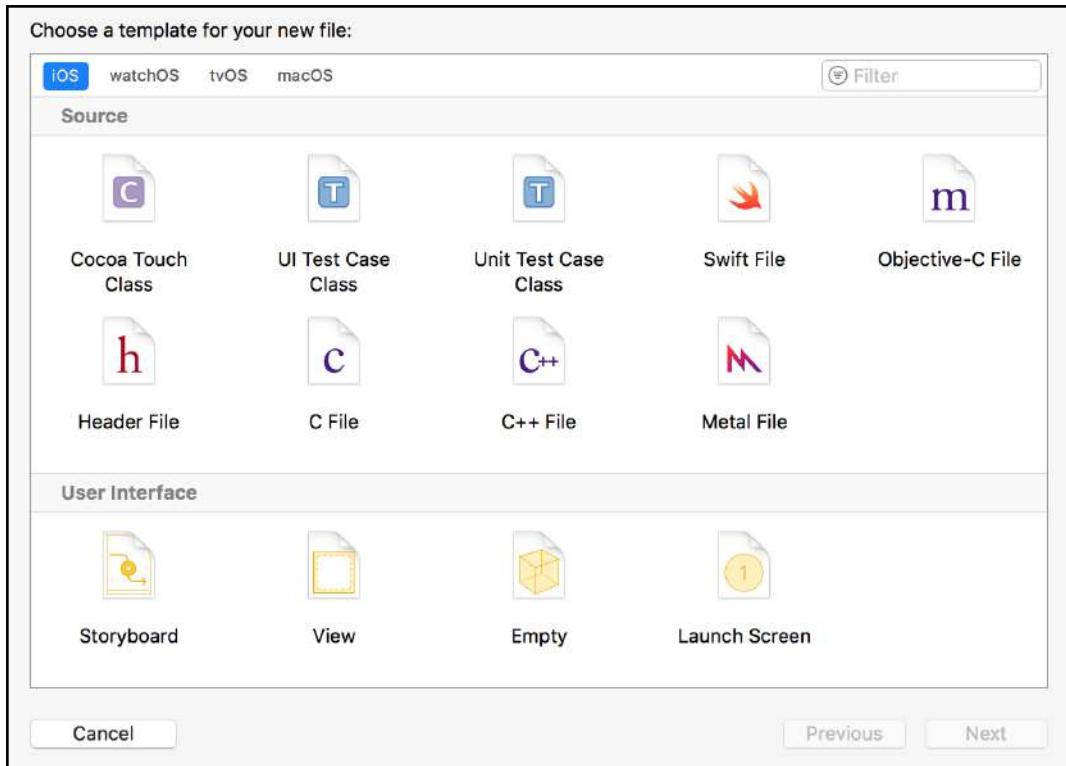


Now that we have covered the basics, let's go back to our app and set up a Collection View using a storyboard. Using Playgrounds is a great way to work things out with your code before coding them in your app.

Creating our controller

Now that we better understand how to create UICollectionView, let's add one to our project. Open your project and do the following steps:

1. Right-click inside of the LetsEat folder and select **New File**.
2. Inside of the **Choose a template for your new file** screen, select **iOS** at the top, and then **Cocoa Touch Class**. Then, hit **Next**:



3. You should see an options screen. Add the following:

New file:

- **Class:** ExploreViewController
- **Subclass:** UIViewController
- **Also create XIB:** Unchecked
- **Language:** Swift

4. Upon hitting **Next**, you are asked to create this file. Select **Create** and then your file should look like mine.

Let's review this `ExploreViewController` class file and also do some maintenance inside of the file. We created this file to use with the `UIViewController` that we created when we initially set up our UI.

Note that there are three methods in this file: `viewDidLoad()`, `didReceiveMemoryWarning()`, and `prepare()` (which is commented out). Let's delete both `didReceiveMemoryWarning()` and `prepare()`, as we do not need them at this time:

The screenshot shows a portion of an Xcode code editor. A red rectangular selection highlights the `didReceiveMemoryWarning()` and `prepare()` methods. To the right of this selection is a red circle with a white 'X' and the word "DELETE". Below this, another section of code is shown with a green circle containing a white '+' and the word "UPDATED".

```
import UIKit

class ExploreViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view. ✖ DELETE
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    /*
    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destinationViewController.
        // Pass the selected object to the new view controller.
    }
*/

}

import UIKit

class ExploreViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

What remains is `viewDidLoad()`; this method is called only once during the life of the View Controller. Let's see what this means by updating `viewDidLoad()` to the following:

```
func viewDidLoad() {
    super.viewDidLoad()
    print("Hello Explore View Controller")
}
```

Now, run the project by hitting the Play button (or using *command + R*). You should now only see `Hello Explore View Controller` inside of the Debug panel.

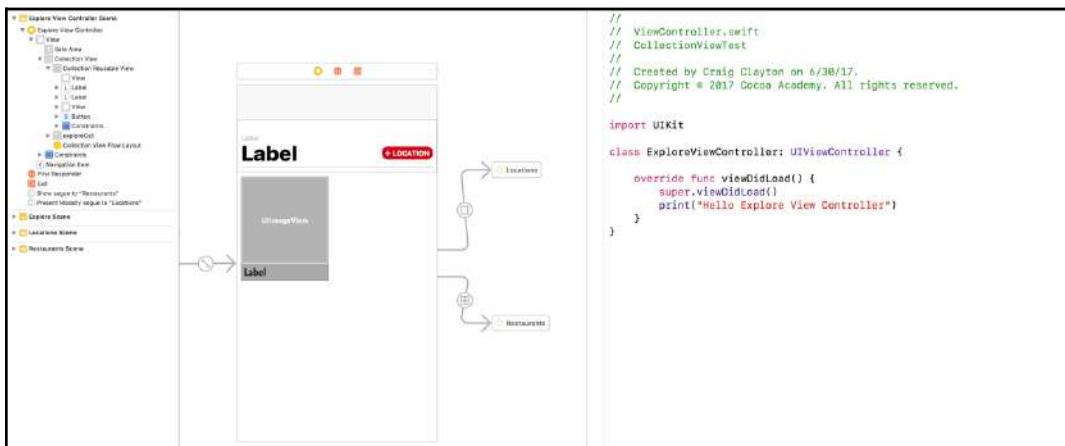
Understanding Collection View controllers and Collection View cells

As noted earlier in the book, Collection View Controllers allow us to display our data in a grid. The individual items inside of Collection Views are called cells, and these cells are what show our data. This data can be anything from an image to text, or both an image and text. You have complete control over what your Collection View cell can display. Our Collection View Controller is responsible for making sure the correct number of cells is displayed.

Let's now connect our file, `ExploreViewController`, with our `UICollectionView` in the storyboard. To do this, we use the Assistant Editor (or split screen), which we access by doing the following:

1. Open `Explore.storyboard`.
2. Close the Navigator panel using the hide Navigator toggle or *command + O*.
3. Close the Utilities panel by hitting the Utilities toggle or use *command + alt + O*.
4. Select the Assistant Editor or use *command + alt + enter*.

You should now see `Explore.storyboard` on the left and `ExploreViewController.swift` on the right:



5. Add the following inside of your `ExploreViewController.swift` file on the line after the following code:

```
class ExploreViewController: UIViewController {  
    @IBOutlet weak var collectionView: UICollectionView!
```



`IBOutlet` is a way to connect to the UI element. We have a Collection View on our `UIViewController`; now, we are creating a variable that allows us to hook into it.

6. After you create the variable, you should see a small circle to the left of the variable:



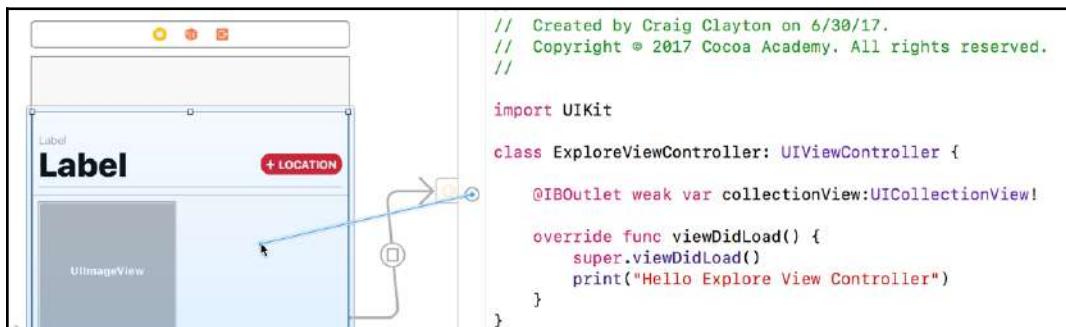
```
@IBOutlet weak var collectionView:UICollectionView!
```

7. When you hover over it, you should see a plus button appear inside of the circle:



```
@IBOutlet weak var collectionView:UICollectionView!
```

Click on it and drag this to your Collection View inside of your `UIViewController`:



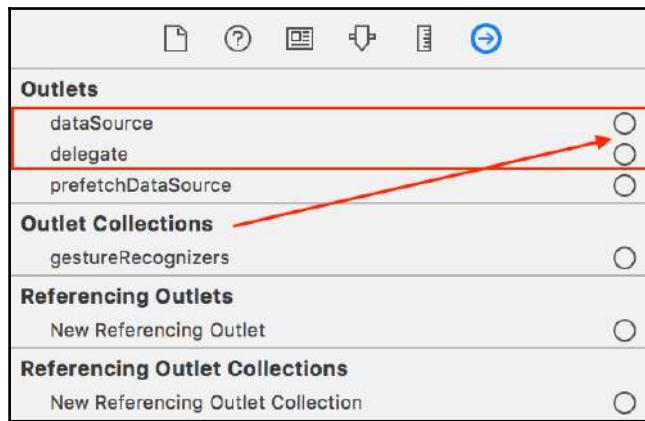
8. Once you release the mouse button, you should see the circle become filled:



```
@IBOutlet weak var collectionView:UICollectionView!
```

9. Select the Standard Editor or use *command + enter*.

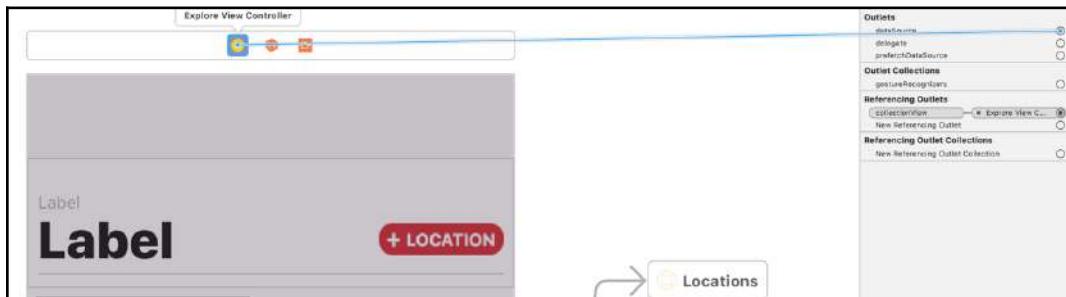
In your scene, select your Collection View. Then, in your Utilities Panel, select the Connections inspector, which is the last icon on the right. Under the **Outlets** section, we now add back `dataSource` and `delegate`, the same ones we removed earlier:



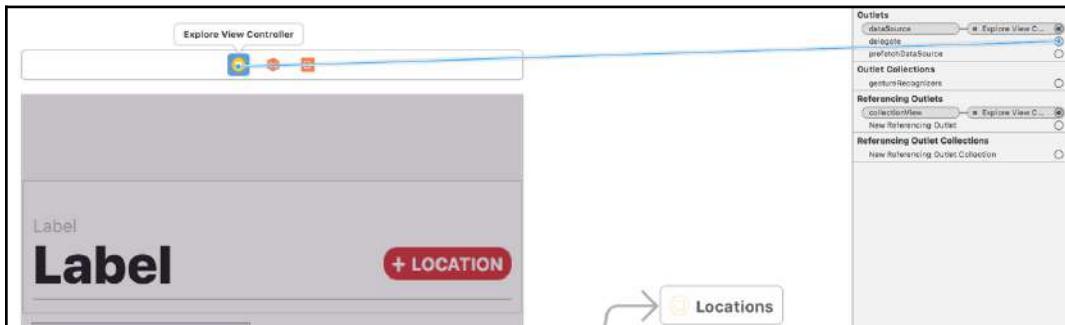
The `dataSource` property is what is used to supply the data for our Collection View, so we need to pass whatever data we have to this property. On the other hand, the `delegate` property, which supplies the behavior, does not require us to supply anything as it receives interactions that happen within our Collection View.

We need to update our data source for our Collection View; let's add this now:

10. Click and drag the `dataSource` property to the Explore View Controller in your Outline view:



11. Click and drag the delegate property to the Explore View Controller in your Outline view:



Getting data into Collection View

Having boxes is great, but having data with beautiful pictures is so much more appealing. Let's get some data displaying inside of our Collection View:

1. Use *command + shift + O*, which opens a small window called **Open Quickly**. Inside the window, type `ExploreView` and hit *enter* to select the `ExploreViewController.swift` file.
2. Update our class definition from the `ExploreViewController: UIViewController` class to the following:

```
class ExploreViewController: UIViewController,  
UICollectionViewDataSource
```

Understanding the data source

Whenever we use Collection View to get data, we must conform to a protocol. A protocol is a set of methods to which we have access, and can either be required or optional. For Collection Views, we are required to implement three methods to get data into a Collection View. So, let's add the following four functions (each beginning with `func`) after the closing curly bracket of `viewDidLoad()`:

```
import UIKit

class ExploreViewController: UIViewController, UICollectionViewDataSource, UICollectionViewDelegate {

    @IBOutlet weak var collectionView: UICollectionView!

    override func viewDidLoad() {
        super.viewDidLoad()
        print("Hello Explore View Controller")
    }

    func collectionView(_ collectionView: UICollectionView, viewForSupplementaryElementOfKind kind: String, at indexPath: IndexPath) -> UICollectionViewCell {
        let headerView = collectionView.dequeueReusableCellSupplementaryView(ofKind: kind, withReuseIdentifier: "header", for: indexPath)
        return headerView
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        return collectionView.dequeueReusableCell(withReuseIdentifier: "exploreCell", for: indexPath)
    }

    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return 1
    }

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return 20
    }

    // Add Unwind here
    @IBAction func unwindLocationCancel(segue: UIStoryboardSegue) {}
}
```

Let's break down the code to better understand what we are doing:

- **Part A:** This first method is what we need to add a header to our Collection View:

```
collectionView(_:viewForSupplementaryElementOfKind:at:)
```

- **Part B:** The identifier is what we added when we were designing in earlier chapters. This identifier helps Xcode know what view we are referring to:

```
let headerView =
collectionView.dequeueReusableCellSupplementaryView(ofKind: kind,
withReuseIdentifier: "header", for: indexPath)
return headerView
```

- **Part C:** Our next method gets called for every item we need. Therefore, in our case, it gets called 20 times:

```
collectionView(_:cellForItemAt:)
```

- **Part D:** Here, we are creating a cell every time `collectionView(_:numberOfItemsInSection:)` is called. The identifier, `exploreCell`, is the name we gave it in the storyboard; so, this is the cell that is grabbed and used inside of our Collection View:

```
return collectionView.dequeueReusableCell(withIdentifier:  
    "exploreCell", for: indexPath)
```

- **Part E:** This method tells our Collection View how many different sections we want to display:

```
numberOfSections(in collectionView: UICollectionView)
```

- **Part F:** Here, we are telling our Collection View that we only want one section:

```
return 1
```

- **Part G:** Our next method tells our Collection View how many different items we are going to display inside of the section we set up:

```
collectionView(_:numberOfItemsInSection:)
```

- **Part H:** We are telling our Collection View that we want to display 20 items:

```
return 20
```

- **Part I:** Finally, we add this line back as it was removed. We use this function to dismiss our location modal when you hit the **Cancel** button:

```
@IBAction func unwindLocationCancel(segue:UIStoryboardSegue)  
{}
```

Let's build and run the project by hitting the Play button (or using *command + R*). We are now finished.

Summary

In this chapter, we covered quite a few new topics, as well as a lot of code. As long as you have a basic understanding of what we covered in this chapter, you will be OK to continue. A lot of these concepts and ideas will be covered again, as these are common design patterns in iOS.

We learned about the MVC architecture. Then, we covered classes and structures, along with their similarities and differences. Finally, we looked at Controllers and classes and how they work. We then created our Controller for our storyboard file.

In the next chapter, we will look at how to add Core Data to an application. We will also look at how to model data in the model editor.

12

Adding Core Data to Your App

Core Data is Apple's data persistence framework. You can utilize this framework whenever your application needs to store data. Simple data can often be stored in `User Defaults`, but when you're handling data that's more complex, has relationships, or needs some form of efficient searching, Core Data is much better suited to your needs.

You don't need to build a very complex app or have vast amounts of data to make Core Data worth your while. Regardless of your app's size, even if it's tiny with only a couple of records, or if you're holding onto thousands of records, Core Data has your back.

In this chapter, you'll learn how to add Core Data to an existing app. The app you will build keeps track of a list of favorite movies for all members of a family. The main interface is a table view that shows a list of family members. If you tap on a family member, you'll see their favorite movies. Adding family members can be done through the overview screen, and adding movies can be done through the detail screen.

You won't build the screens in this app from scratch. The code bundle for this chapter includes a starter project called **MustC**. The starter project contains all of the screens, so you don't have to set up the user interface before you get around to implementing Core Data.

In this chapter, the following topics are covered:

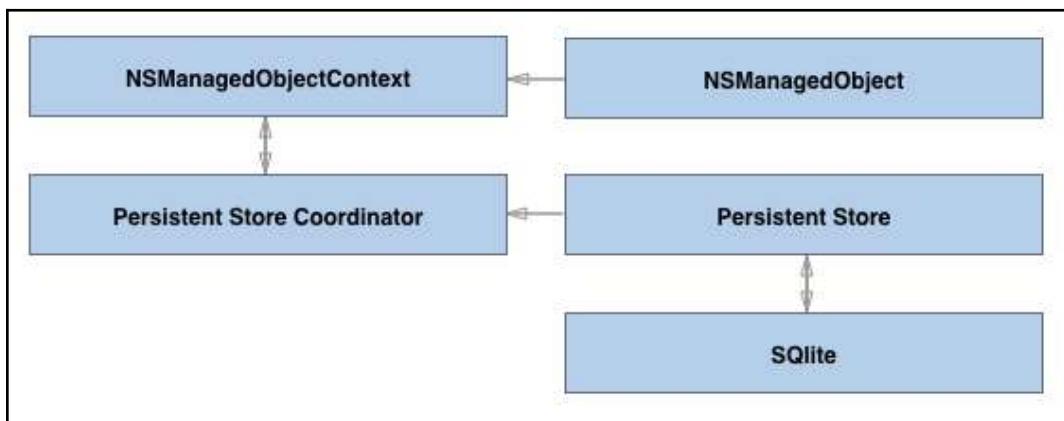
- Understanding the Core Data Stack
- Adding Core Data to an application
- Modeling data in the model editor

- Storing and querying data
- Using multiple managed object contexts

Understanding the Core Data Stack

Before you dive right into the project and add Core Data to it, let's take a look at how Core Data actually works, what it is, and what it isn't. In order to make efficient use of Core Data, it's essential that you know what you're working with.

When you work with Core Data, you're actually utilizing a stack of layers that starts with managed objects and ends with a data store. This is often a SQLite database, but there are different storage options you can use with Core Data, depending on your application needs. Let's take a quick look at the layers involved with Core Data and discuss their roles in an application briefly:



At the top-right of this diagram is the **NSManagedObject** class. When you use Core Data, this is the object you'll interact with most often since it's the base class for all Core Data models your app contains. For instance, in the app you will build in this chapter, the family member and movie models are subclasses of **NSManagedObject**.

Each managed object belongs to an **NSManagedObjectContext**. The managed object context is responsible for communicating with the persistent store coordinator. Often, you'll only need a single managed object context and a single Persistent Store Coordinator. However, it is possible to use multiple persistent store coordinators and multiple managed object contexts. It's even possible to have multiple managed object contexts for the same persistent store coordinator.

A setup with multiple managed object contexts can be particularly useful if you're performing costly operations on your managed objects; for example, if you're importing or synchronizing large amounts of data. Usually, you will stick to using a single managed object context and a single persistent store coordinator because most apps don't need more than one.

The persistent store coordinator is responsible for communicating with the persistent store. In most scenarios, the persistent store uses SQLite as its underlying storage database. However, you can also use other types of storage, such as an in-memory database. An in-memory database is especially useful if you're writing unit tests or if your app has no need for long-term storage.

If you've worked with MySQL, SQLite, or any other relational database, it is tempting to think of Core Data as a layer on top of a relational database. Although this isn't entirely false since Core Data can use SQLite as its underlying storage, Core Data does not work the same as using SQLite directly; it's an abstraction on top of this.

One example of a difference between SQLite and Core Data is the concept of primary keys. Core Data doesn't allow you to specify your own primary keys. Also, when you define relationships, you don't use foreign keys. Instead, you simply define the relationship and Core Data will figure out how to store this relationship in the underlying database. You will learn more about this later. It's important to know that you should not directly translate your SQL experiences to Core Data. If you do, you will run into issues, simply because Core Data is not SQL. It just so happens that SQLite is one of the ways that data can be stored but the similarities really do end right there.

To recap, all core data apps have a **persistent store**. This store is backed by an in-memory database or a SQLite database. A **persistent store coordinator** is responsible for communicating with the **persistent store**. The object communicating with the **persistent store coordinator** is the **managed object context**. An application can have multiple **managed object context** instances talking to the same **persistent store coordinator**. The objects that a **managed object context** retrieves from the **persistent store coordinator** are **managed objects**.

Now that you have an overview of the Core Data stack and where all the parts involved with its usage belong, let's add the Core Data stack to the **MustC** application.

Adding Core Data to an existing application

When you create a new project in Xcode, Xcode asks whether you want to add Core Data to your application. If you check this checkbox, Xcode will automatically generate some boilerplate code that sets up the Core Data stack. For practicing purposes, **MustC** was set up without Core Data so you'll have to add this to the project yourself. Start by opening `AppDelegate.swift` and add the following import statement:

```
import CoreData
```

Next, add the following `lazy` variable to the implementation of `AppDelegate`:

```
private lazy var persistentContainer: NSPersistentContainer = {
    let container = NSPersistentContainer(name: "MustC")
    container.loadPersistentStores(completionHandler: {
        (storeDescription, error) in
        if let error = error {
            fatalError("Unresolved error (error), (error.userInfo)")
        }
    })
    return container
}()
```



If you declare a variable as `lazy`, it won't be initialized until it is accessed. This is particularly useful for variables that are expensive to initialize, rely on other objects, or are not always accessed. The fact that the variable is initialized just in time comes with a performance penalty since the variable needs to be set up the first time you access it. In certain cases this is fine, but in other cases, it might negatively impact the user experience. When used correctly, `lazy` variables can offer great benefits.

The preceding code snippet creates an instance of `NSPersistentContainer`. The persistent container is a container for the persistent store coordinator, persistent store, and managed object context. This single object manages different parts of the Core Data stack, and it ensures that everything is set up and managed correctly.

If you let Xcode generate the Core Data code for your app, it adds a similar property to create an `NSPersistentContainer`. Xcode also adds a method called `saveContext()` to `AppDelegate`. This method is used in `applicationWillTerminate(_:)` to perform a last-minute save of changes and updates when the application is about to terminate. Since you're setting up Core Data manually, this behavior isn't added by Xcode so it must be added by you manually.

Instead of placing the `saveContext()` method in `AppDelegate`, you will add this method as an extension to `NSPersistentContainer`. This makes it easier for other parts of your code to use this method, without having to rely on `AppDelegate`.

Create a new folder in the **Project Navigator** and name it `Extensions`. Also, create a new Swift file and name it `NSPersistentContainer.swift`. Add the following implementation to this file:

```
import CoreData

extension NSPersistentContainer {
    func saveContextIfNeeded() {
        if viewContext.hasChanges {
            do {
                try viewContext.save()
            } catch {
                let nserror = error as NSError
                fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
            }
        }
    }
}
```

This code adds a new method to `NSPersistentContainer` instances by extending it. This is really convenient because it decouples the `save` method from `AppDelegate` entirely. This is much nicer than the default save mechanism provided for Core Data apps by Xcode. Add the following implementation of `applicationWillTerminate(_:)` to `AppDelegate` to save the context right before the app terminates:

```
func applicationWillTerminate(_ application: UIApplication) {
    persistentContainer.saveContextIfNeeded()
}
```

Now, whenever the application terminates, the persistent store will check whether there are any changes to the managed object context that the `viewContext` property points to. If there are any changes, an attempt to save them is made. If this attempt fails, the app will crash with a `fatalError`. In your own app, you might want to handle this scenario a bit more gracefully. It could very well be that failing to save data before the app terminates is no reason to crash in your app. You can modify the error-handling implementation of `saveContextIfNeeded()` if you think a different behavior is more appropriate for your app.

Now that you have the Core Data stack set up, you need a way to provide this stack to the view controllers in the app. A common technique to achieve this is called *dependency injection*. In this case, dependency injection means that `AppDelegate` will pass the persistent container to `FamilyMemberViewController`, which is the first view controller in the app. It then becomes the job of `FamilyMemberViewController` to pass the persistent container to the next view controller that depends on it, and so forth.

In order to inject the persistent container, you need to add a property to `FamilyMembersViewController` that holds the persistent container. Don't forget to import Core Data at the top of the file and add the following code:

```
var persistentContainer: NSPersistentContainer!
```

Now, in `AppDelegate`, modify the `application(_: didFinishLaunchingWithOptions:)` method as follows:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    if let navVC = window?.rootViewController as?
    UINavigationController,
        let initialVC = navVC.viewControllers[0] as?
    FamilyMembersViewController {
        initialVC.persistentContainer = persistentContainer
    }
    return true
}
```

Even though this code does exactly what it should, you can make one major improvement. You know that there might be more view controllers that depend on a persistent container. You also learned that checking whether something is a certain type is something you should generally avoid. As an exercise, attempt to improve the code by adding a protocol called `PersistentContainerRequiring`. This protocol should add a requirement for an implicitly-unwrapped `persistentContainer` property. Make sure that `FamilyMembersViewController` conforms to this protocol and fix the implementation of `application(_: didFinishLaunchingWithOptions:)` as well so it uses your new protocol.

You have just put down the foundation that is required to use Core Data in your app. Before you can use Core Data and store data in it, you must define what data you would like to save by creating your data model. Let's go over how to do this next.

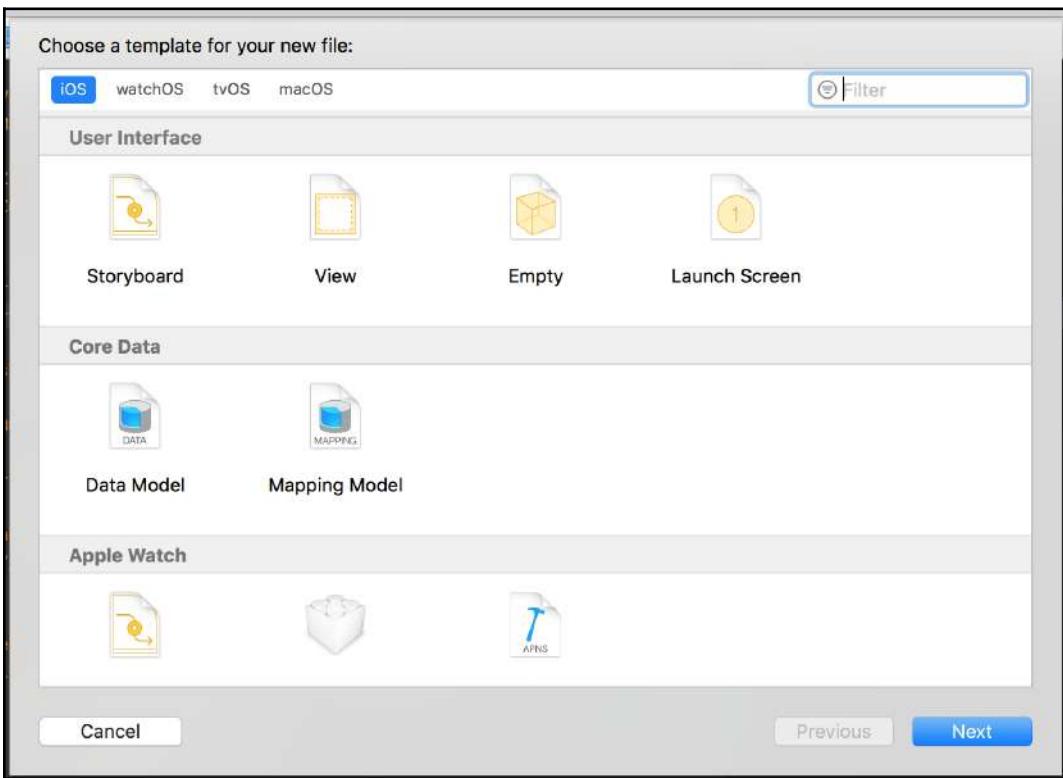
Creating a Core Data model

So far, you have worked on the persistence layer of your app. The next step is to create your models so you can actually store and retrieve data from your Core Data database. All models in an application that uses Core Data are represented by `NSManagedObject` subclasses. When you retrieve data from your database, `NSManagedObjectContext` is responsible for creating instances of your managed objects and populating them with the relevant fetched data.

The **MustC** application requires two models: a family-member model and a movie model. When you define models, you can also define relationships. For the models in **MustC**, you should define a relationship that links multiple movies to a single family member.

Creating the models

In order for Core Data to understand which models your application uses, you must define them in Xcode's model editor. Let's create a new model file so you can add your own models to the **MustC** application. Create a new file, and from the file template selection screen, pick **Data Model**. Name your model file `MustC`. First, you will set up the basic models, and then see how you can define a relationship between family members and their favorite movies:



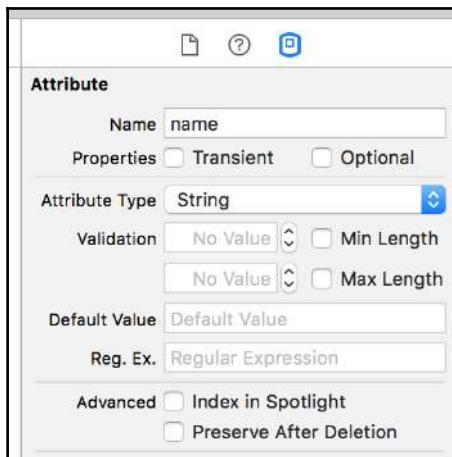
Your project now contains a file called `MustC.xcdatamodeld`. Open this file to go to the model editor. In the bottom-left corner of the editor, you'll find a button labeled **Add Entity**. Click this button to add a new Entity and name it `FamilyMember`.

When you select an Entity by clicking it, you can see all of its attributes, relationships, and fetched properties. Let's add a **name** property to the family member. Click on the plus (+) icon at the bottom of the empty attributes list and add a new attribute called **name**. Make sure that you select **String** as the type for this attribute:



Click on this new property to select it. In the sidebar on the right, select the third tab to open the **Data Model inspector**. This is where you can see more detailed information on this attribute. For instance, you can configure a property to be indexed for faster lookups. You can also choose whether you want the attribute to be optional. For now, you shouldn't care too much about indexing since you're not performing lookups by family members' names, and, even if you were, a family doesn't tend to have hundreds or thousands of members. By default, the **Optional** checkbox is checked. Make sure that you uncheck this box because you don't want to store family members without a name.

Some other options you have for attributes are adding validation, adding a default value, and enabling indexing in Spotlight. For now, leave all those options in their default setting:



In addition to a `FamilyMember` Entity, `MustC` also needs a `Movie` Entity. Create this entity using the same steps as before and give it a single property: `title`. This property should be a string and it shouldn't be optional. Once you've added this property, you can set up a relationship between family members and their favorite movies.

Defining relationships

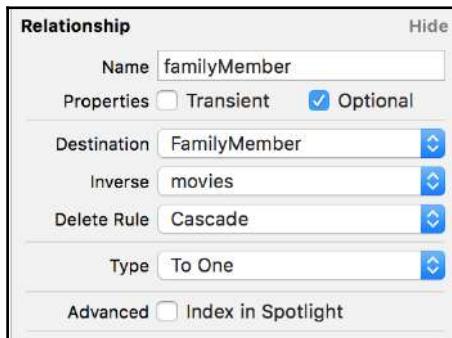
A relationship in Core Data adds a reference to an Entity as a property on an Entity. In this case, you want to define a relationship between `FamilyMember` and `Movie`.

The best way to describe this relationship is a one-to-many relationship. This means that every movie will have only one family member associated with it and every family member can have multiple favorite movies.



Configuring your data model with a one-to-many relationship from `Movie` to `FamilyMember` is not the most efficient way to define this relationship. A many-to-many relationship is likely a better fit because that would allow multiple family members to add the same movie instance as their favorite. A one-to-many relationship is used in this example to keep the setup simple and make it easy to follow along with the example.

Select the `FamilyMember` entity and click on the plus icon at the bottom of the **Relationships** list. Name the relationship `movies` and select `Movie` as the destination. Don't select an **Inverse** relationship yet because the other end of this relationship is not defined yet. The **Inverse** relationship will tell the model that `Movie` has a property that points back to the `FamilyMember`. Make sure that you select **to many** as the relationship type in the **Data Model Inspector** panel for the `movies` property. Also, select **Cascade** as the value for the **delete rule**:



The **delete rule** is a very important property to be set correctly. Not paying attention to this property could result in a lot of orphaned, and even corrupted, data in your database. For instance, setting this property to `nullify` simply sets the **Inverse** of the relationship to `nil`. This is the correct behavior when deleting a movie because deleting a movie shouldn't delete the entire family member who added this movie as their favorite. It should simply be removed from the list of favorites.

However, if a `FamilyMember` gets deleted and the relationship is nullified, you would end up with a bunch of movies that don't have a family member associated with them. In this application, these movies are worthless; they won't be used anymore because every movie only belongs to a single `FamilyMember`. For this app, it's desirable that when a `FamilyMember` gets deleted, Core Data also deletes their favorite movies. This is precisely what the cascade option does; it cascades the deletion over to the relationship's **Inverse**.

After setting the delete rule to cascade, select the `Movie` entity and define a relationship called `familyMember`. The destination should be `FamilyMember` and the **Inverse** for this relationship is `favoriteMovies`. After adding this relationship, the **Inverse** will be automatically set on the `FamilyMember` entity:



Using your entities

As mentioned before, every model or Entity in your Core Data database is represented by an `NSManagedObject`. There are a couple of ways to create or generate `NSManagedObject` subclasses. In the simplest of setups, an `NSManagedObject` subclass contains just the properties for a certain managed object and nothing else. If this is the case, you can let Xcode generate your model classes for you.

This is actually what Xcode does by default. If you build your project now and add the following code to `viewDidLoad()` in `FamilyMembersViewController`, your project should compile just fine:

```
let fam = FamilyMember(entity: FamilyMember.entity(), insertInto:  
persistentContainer.viewContext)
```

This works automatically; you don't have to write any code for your models yourself. Don't worry about what the preceding code does just yet, we'll get into that very soon. The point is that you see that a `FamilyMember` class exists in your project even though you didn't have to create one yourself.

If the default behavior doesn't suit the approach you want in your app – for instance, if you want to prevent your code from modifying your models by defining your variables as `private(set)` – you may want to create a custom subclass instead of making Xcode generate the classes for you. A custom `NSManagedObject` subclass for `FamilyMember` could look like this:

```
class FamilyMember: NSManagedObject {  
    @NSManaged private(set) var name: String  
    @NSManaged private(set) var favoriteMovies: [Movie]?  
}
```

This custom `FamilyMember` subclass makes sure that external code can't modify the instances by making the setters on `FamilyMember` private. Depending on your application, it might be a good idea to implement this since it will ensure that your models can't accidentally change.

One final option you have is to let Xcode generate the properties for your `NSManagedObject` as an extension on a class you define. This is particularly useful if you have some custom stored properties that you'd like to define on your model or if you have a customized `NSManagedObject` subclass that you can use as the base for all of your models.



All code that Xcode generates for your Core Data models is added to the `Build` folder in Xcode's Derived Data. You shouldn't modify it, or access it directly. These files will be automatically regenerated by Xcode whenever you perform a build, so any functionality you add inside the generated files will be overwritten.

For the **MustC** app, it's okay if Xcode generates the model definition classes since there are no custom properties that you need to add. In the model editor, select each entity and make sure that the **Codegen** field is set to **Class Definition**; you can find this field in the Data Model inspector panel:



At this point, you are all set up to store your first data in the Core Data database.

Reading and writing data in a Core Data database

The first step to implement data persistence for your app is to make sure that you can store data in the database. You have defined the models that you want to store in your database so the next step is to actually store your models. Once you have implemented a rough version of your data persistence, you will refine the code to make it more reusable. The final step will be to read data from Core Data and dynamically respond to changes in the database.

Understanding data persistence

Whenever you want to persist a model with Core Data, you must insert a new `NSManagedObject` into an `NSManagedObjectContext`. Doing this does not immediately persist the model. It merely stages the object for persistence in the current `NSManagedObjectContext`. If you don't properly manage your managed objects and contexts, this is a potential source of bugs. For example, not persisting your managed objects results in the loss of your data once you refresh the context. Even though this might sound obvious, it could lead to several hours of frustration if you aren't aware of this and have bugs in managing your managed object context.

If you want to save managed objects correctly, you must tell the managed-object context to persist its changes to the persistent store coordinator. The persistent store coordinator will take care of persisting the data in the underlying SQLite database.

Extra care is required when you use multiple managed object contexts. If you insert an object in one managed object context and persist it, you will manually need to synchronize the changes into the other managed object contexts. Also, managed objects are not thread-safe. This means that you must make sure that you create, access, and store a managed object on a single thread at all times. The managed-object context has a helper method called `perform(_:)` to help you with this.

Inserting new objects, updating them, or adding relationships between objects should always be done using the `perform(_:)` method. The reason is that the helper method makes sure that all code in the closure you want to perform are executed on the same thread that the managed object context is on.

Now that you're aware of how data persistence works in Core Data, it's time to start implementing the code to store family members and their favorite movies. You will implement the family member persistence first. Then you'll expand the app so you can safely add movies to family members.

Persisting your models

The first model you will persist is the family member model. The app is already set up with a form that asks for a family member name and a delegate protocol that informs `FamilyMembersViewController` whenever the user wants to store a new family member.

Note that none of the input data is validated; usually, you'd want to add some checks that make sure that the user is not trying to insert an empty family member name, for instance. For now, we'll skip that because this type of validation isn't Core-Data-specific.

The code to persist new family members should be added to the `saveFamilyMember(withName:)` method. Add the following implementation to `FamilyMembersViewController`; we'll go over it line by line after adding the code:

```
func saveFamilyMember(withName name: String) {
    // 1
    let moc = persistentContainer.viewContext

    // 2
    moc.perform {
        // 3
        let familyMember = FamilyMember(context: moc)
        familyMember.name = name

        // 4
        do {
            try moc.save()
        } catch {
            moc.rollback()
        }
    }
}
```

The first comment in this code marks where the managed object context is extracted from `persistentContainer`. All `NSPersistentContainer` objects have a `viewContext` property. This property is used to obtain a managed object context that exists on the main thread.

The second comment marks the call to `perform(_:_)`. This ensures that the new `FamilyMember` instance is created and stored on the correct thread. When you create an instance of a managed object context, you must pass the managed object context to the instance of `NSManagedObject`.

Lastly, saving the managed object context can fail, so you must wrap the call to `save()` in a `do {} catch {}` block, so it correctly handles potential errors. If the managed object context can't be saved, all unsaved changes are rolled back.

This code is all you need to persist family members. Before you implement the required code to read existing family members and respond to the insertion of new family members, let's set up `MoviesViewController` so it can store movies for a family member.

The code to store movies for a family member is very similar to the code you wrote earlier. Before you implement the following snippets, make sure that you conform `MoviesViewController` to `PersistentContainerRequiring` and import `CoreData`. Also, add a `persistentContainer` property to `MoviesViewController`.

In order to connect a new movie to a family member, you also need a variable to hold the family member in `MoviesViewController`. Add the following declaration to `MoviesViewController`:

```
var familyMember: FamilyMember?
```

After doing this, add the following implementation for `saveMovie(withName:)`:

```
func saveMovie(withName name: String) {
    guard let familyMember = self.familyMember
    else { return }

    let moc = persistentContainer.viewContext

    moc.perform {
        let movie = Movie(context: moc)
        movie.title = name

        // 1
        let newFavorites: Set<AnyHashable> =
```

```
familyMember.movies?.adding(movie) ?? [movie]

// 2
familyMember.movies = NSSet(set: newFavorites)

do {
    try moc.save()
} catch {
    moc.rollback()
}
}
```

The most important differences between adding the movie and the family member are highlighted with comments. Note that the `movies` property on a family member is `NSSet`. This is an immutable object so you need to create a copy and add the movie to that copy. If no copy could be made because there is no set created yet, you can create a new set with the new movie in it. Next, this new updated `Set` is converted back to an `NSSet` so it can be the new value for `movies`.

As you can see, both `save` methods share about half of the implementation. You can make some clever use of extensions and generics in Swift to avoid writing this duplicated code. Let's refactor the app a bit.

Refactoring the persistence code

Many iOS developers dislike the amount of boilerplate code that is involved with using Core Data. Simply persisting an object requires you to repeat several lines of code, which can become quite a pain to write and maintain over time. The approach to refactoring the persistence code presented in the following examples is heavily inspired by the approach taken in the *Core Data* book written by *Florian Kugler* and *Daniel Eggert*.



If you're interested in learning more about Core Data outside of what this book covers, and if you'd like to see more clever ways to reduce the amount of boilerplate code, you should pick up this book.

One pattern you can find in both `save` methods is the following:

```
moc.perform {
    // create managed object

    do {
```

```
    try moc.save()
} catch {
    moc.rollback()
}
}
```

It would be great if you could write the following code to persist data instead:

```
moc.persist {
    // create managed object
}
```

This can be achieved by writing an extension for `NSManagedObjectContext`. Add a file called `NSManagedObjectContext` to the extensions folder, and add the following implementation:

```
import CoreData

extension NSManagedObjectContext {
    func persist(block: @escaping () -> Void) {
        perform {
            block()

            do {
                try self.save()
            } catch {
                self.rollback()
            }
        }
    }
}
```

The preceding code allows you to reduce the amount of boilerplate code, which is something that you should always try to achieve. Reducing boilerplate code greatly improves your code's readability and maintainability. Update both the family overview and the movie list view controllers to make use of this new persistence method.

Reading data with a simple fetch request

The simplest way to fetch data from your database is to use a fetch request. The managed object context forwards fetch requests to the persistent store coordinator. The persistent store coordinator will then forward the request to the persistent store, which will then convert the request to a SQLite query. Once the results are fetched, they are passed back up this chain and converted to `NSManagedObject` instances.

By default, these objects are called faults. When an object is a fault, it means that the actual properties and values for the object are not fetched yet, but they will be fetched once you access them. This is an example of a good implementation of lazy variables because fetching the values is a pretty fast operation, and fetching everything up front would greatly increase your app's memory footprint because all values would have to be loaded into memory right away.

Let's take a look at an example of a simple fetch request that retrieves all `FamilyMember` instances that were saved to the database:

```
let request: NSFetchedResultsController<FamilyMember> =
    FamilyMember.fetchRequest()

let moc = persistentContainer.viewContext

guard let results = try? moc.fetch(request)
    else { return }
```

As you can see, it's not particularly hard to fetch all of your family members. Every `NSManagedObject` has a class method that configures a basic fetch request that can be used to retrieve data. If you have large amounts of data, you probably don't want to fetch all of the persisted objects at once. You can configure your fetch request to fetch data in batches by setting the `fetchBatchSize` property. It's recommended that you use this property whenever you want to use fetched data in a table view or collection view. You should set the `fetchBatchSize` property to a value that is just a bit higher than the number of cells you expect to display at a time. This makes sure that Core Data fetches plenty of items to display while avoiding loading everything at once.

Now that you know how to fetch data, let's display some data in the family member's table view. Add a new variable called `familyMembers` to `FamilyMembersViewController`. Give this property an initial value of `[FamilyMember]()`, so you start off with an empty array of family members. Also, add the example fetch request you saw earlier to `viewDidLoad()`. Next, assign the result of the fetch request to `familyMembers` as follows:

```
familyMembers = results
```

Finally, update the table view delegate methods so `tableView(_:numberOfRowsInSection:)` returns the number of items in the `familyMembers` array. Also, update the `tableView(_:cellForRowAtIndexPath:)` method by adding the following two lines before returning the cell:

```
let familyMember = familyMembers[indexPath.row]
cell.textLabel?.text = familyMember.name
```

If you build and run your app now, you should see the family members you already saved. New family members won't show up right away. However, when you quit the app and run it again, new members will show up.

You could manually reload the table view right after you insert a new family member so it's always up to date but this isn't the best approach. You will see a better way to react to the insertion of new data soon. Let's finish the family member detail view first so it shows a family member's favorite movies. Add the following code to the end of the `prepare(for:sender:)` method in the overview view controller:

```
if let moviesVC = segue.destination as? MoviesViewController {
    moviesVC.persistentContainer = persistentContainer
    moviesVC.familyMember = familyMembers[selectedIndex.row]
}
```

The preceding lines of code pass the selected family member and the persistent container to `MoviesViewController` so it can display and store the current family member's favorite movies.

All you need to do to show the correct movies for a family member is use the family member's favorite movies in the `MovieViewController` class's table-view data-source methods, as follows:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return familyMember?.movies?.count ?? 0
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    guard let cell = tableView.dequeueReusableCell(withIdentifier: "MovieCell"),
          let movies = familyMember?.movies
    else { fatalError("Wrong cell identifier requested or missing family member") }

    let moviesArray = Array(movies as! Set<Movie>)
    let movie = moviesArray[indexPath.row]
    cell.textLabel?.text = movie.title

    return cell
}
```

You don't need to use a fetch request here because you can simply traverse the `movies` relationship on the family member to get their favorite movies. This isn't just convenient for you as a developer, it's also good for your app's performance. Every time you use a fetch request, you force a query to the database. If you traverse a relationship, Core Data will attempt to fetch the object from memory instead of asking the database.

Again, adding new data won't immediately trigger the table view to update its contents. We'll get to that after we take a look at how to filter data. If you want to check whether your code works, build and rerun the app so all the latest data is fetched from the database.

Filtering data with predicates

A typical operation you'll want to perform on your database is filtering. In Core Data, you use predicates to do this. A **predicate** describes a set of rules that any object that gets fetched has to match.

When you model your data in the model editor, it's wise to think about the types of filtering you need to do. For instance, you may be building a birthday calendar where you'll often sort or filter by date. If this is the case, you should make sure that you have a Core Data index for this property. You can enable indexing with the checkbox you saw earlier in the model editor. If you ask Core Data to index a property, it will significantly improve performance when filtering and selecting data in large datasets.

Writing predicates can be confusing, especially if you try to think of them as the `where` clause from SQL. Predicates are very similar, but they're not quite the same. A simple predicate looks as follows:

```
NSPredicate(format: "name CONTAINS [n] %@", "Gu")
```

A predicate has a format; this format always starts with a key. This key represents the property you want to match on. In this example, it would be the name of a family member. Then, you specify the condition, for instance `==`, `>`, `<`, or `CONTAINS [n]`.

There are more conditions available, but the ones listed are some examples of conditions you'll commonly use. Finally, you will specify a placeholder that is substituted with the true value. This placeholder is `%@` in the preceding sample. If you have written any Objective-C before you picked up this book, the `%@` placeholder might look familiar to you because it's used as a placeholder in format strings there.

The example predicate is very simple and bare; it could be the template for a search feature you're building. Usually, a simple search doesn't have to be much more complicated than this as long as there's an index added to the properties you search for.

If you have multiple predicates you want to match on, you can combine them using `NSCompoundPredicate`. This class combines different predicates using either an `and`, `or`, or `not` clause. A typical use case for this approach is when you build a complex filter in your app where the predicate is hard to express in a single statement.

To use a predicate in a fetch request, you assign it to the `predicate` property of a fetch request. Every fetch request has a `predicate` property that you can set. It can handle both a single predicate and a compound predicate. If you set this property before executing the fetch request, the predicate is applied to the request, and you will receive a filtered dataset instead of the full dataset.

Predicates are powerful, and they have many options available.



If you're interested in an in-depth overview of predicates and all of the ways in which you can make use of format strings, I recommend that you read *Apple's Predicate Programming Guide* at <http://apple.co/2fF3qHc>. It provides a well-documented overview of predicates and their applications.

Next up, you will learn how to respond to changes in the managed object context. For instance, when you add new family members and movies.

Reacting to database changes

In its current state, the `MustC` app doesn't update its list when a new managed object is persisted. One possible solution for this is to manually reload the table right after a new family member is inserted. Although this might work well for some time, it's not the best solution to this problem. If the app grows, you might add a functionality that imports new family members from the network. Manually refreshing the table view would be problematic because the networking logic should not be aware of the table view. Luckily, there is a better solution to react to changes in your data.

One way to respond to database changes is using `NSFetchedResultsController`. This object is perfect for listening to the insertion of new family members. A second way to respond to notifications is through notifications. You will implement this approach in `MoviesViewController`.

Implementing NSFetchedResultsController

`NSFetchedResultsController` is a helper object that specializes in fetching data and managing this data. It listens to changes in its managed object context and notifies a delegate whenever the data it has fetched changes. This is incredibly helpful because it allows you to respond to specific changes in the dataset rather than reloading the table view entirely.

Being a delegate for the fetched results controller involves the following four important methods:

- `controllerWillChangeContent (_:)`
- `controllerDidChangeContent (_:)`
- `controller (_:didChange:at:for:newIndexPath:)`
- `controller (_:didChange:atSectionIndex:for:)`

The first method, `controllerWillChangeContent (_:)`, is called right before the controller passes updates to the delegate. If you're using a table view with a fetched-results controller, this is the perfect method to begin updating the table view.

Next, `controller (_:didChange:at:for:newIndexPath:)` and `controller (_:didChange:atSectionIndex:for:)` are called to inform the delegate about updates to the fetched items and sections, respectively. This is where you should handle updates in the fetched data. For instance, you could insert new rows in a table view if new items were inserted in the dataset.

Finally, `controllerDidChangeContent (_:)` is called. This is the point where you should let the table view know you've finished processing the updates so all the updates can be applied to the table view's interface.

For **MustC**, it doesn't make sense to implement all four methods because the table view that shows family members only has a single section. This means `controller (_:didChange:atSectionIndex:for:)` does not have to be implemented.

To use a fetched-results controller to fetch the stored family members, you need to create an instance of `NSFetchedResultsController` and assign `FamilyMembersViewController` as its delegate so it can respond to changes in the underlying data. You can then implement the delegate methods so you can respond to changes in the fetched-results dataset. Remove the `familyMembers` array from the variable declarations in `FamilyMembersViewController` and add the following `fetchedResultsController` property:

```
var fetchedResultsController:  
NSFetchedResultsController<FamilyMember>?
```

The `viewDidLoad` method should be adjusted as follows:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    let moc = persistentContainer.viewContext  
    let request = NSFetchedResultsController<FamilyMember>(entityName:  
    "FamilyMember")  
  
    request.sortDescriptors = [NSSortDescriptor(key: "name", ascending:  
    true)]  
    fetchedResultsController = NSFetchedResultsController(fetchRequest:  
request, managedObjectContext: moc, sectionNameKeyPath: nil,  
cacheName: nil)  
  
    fetchedResultsController?.delegate = self  
  
    do {  
        try fetchedResultsController?.performFetch()  
    } catch {  
        print("fetch request failed")  
    }  
}
```

This implementation initializes `NSFetchedResultsController`, assigns its delegate, and tells it to execute the fetch request. Note that the `sortDescriptors` property of the fetch request is set to an array that contains `NSSortDescriptor`. A fetched-request controller requires this property to be set, and for the list of family members, it makes sense to order family members by name.

Now that you have a fetched-results controller, you should implement the delegate methods on `FamilyMembersViewController` and make it conform to `NSFetchedResultsControllerDelegate`. Add the following extension to `FamilyMembersViewController.swift`:

```
extension FamilyMembersViewController:  
NSFetchedResultsControllerDelegate {  
    func controllerWillChangeContent(_ controller:  
    NSFetchedResultsController<NSFetchRequestResult>) {  
        tableView.beginUpdates()  
    }  
  
    func controllerDidChangeContent(_ controller:  
    NSFetchedResultsController<NSFetchRequestResult>) {
```

```
        tableView.endUpdates()
    }
}
```

The implementation for this extension is fairly straightforward. The table view gets notified when the fetched-result controller is about to process changes to its data and when the fetched-results controller is done processing changes. The bulk of the work needs to be done in `controller(_:didChange:at:for:newIndexPath)`. This method is called when an update has been processed by the fetched-result controller. In **MustC**, the goal is to update a table view, but you could also update a collection view or store all of the updates in a list and do something else with them.

Let's take a look at how you can process changes to fetched data in the following method:

```
func controller(_ controller:
NSFetchedResultsController<NSFetchRequestResult>, didChange anObject:
Any, at indexPath: IndexPath?, for type: NSFetchedResultsChangeType,
newIndexPath: IndexPath?) {

    switch type {
    case .insert:
        guard let insertIndex = newIndexPath
        else { return }
        tableView.insertRows(at: [insertIndex], with: .automatic)
    case .delete:
        guard let deleteIndex = indexPath
        else { return }
        tableView.deleteRows(at: [deleteIndex], with: .automatic)
    case .move:
        guard let fromIndex = indexPath,
              let toIndex = newIndexPath
        else { return }
        tableView.moveRow(at: fromIndex, to: toIndex)
    case .update:
        guard let updateIndex = indexPath
        else { return }
        tableView.reloadRows(at: [updateIndex], with: .automatic)
    }
}
```

This method contains quite a lot of code, but it's actually not that complex. The preceding method receives a type parameter. This parameter is an `NSFetchedResultsChangeType` that contains information about the kind of update that was received. The following are the four types of updates that can occur:

- insert
- delete
- move
- update

Each of these change types corresponds to a database action. If an object was inserted, you will receive an `insert` change type. The proper way to handle these updates for **MustC** is to simply pass them on to the table view. Once all updates are received, the table view will apply all of these updates at once.

If you had implemented `controller(_:didChange:atSectionIndex:for:)` as well, it would also have received a change type; however, the sections only deal with the following two types of changes:

- Insert
- Delete

Sections don't update or move, so if you implement this method, you don't have to account for all cases because you won't encounter any, other than the two listed types of changes.

If you take a close look at the implementation for `controller(_:didChange:at:for:newIndexPath)`, you'll notice that it receives two index paths. One is named `indexPath`, and the other is named `newIndexPath`. They're both optional, so you will need to make sure that you safely unwrap them if you use them. For new objects, only the `newIndexPath` property will be present. For delete and update, the `indexPath` property will be set. When an object is moved from one place in the dataset to another, both `newIndexPath` and `indexPath` will have a value.

The last thing you need to do is update the code in `FamilyMembersViewController` so it uses the fetched results controller instead of the `familyMembers` array that it used earlier. First, update the `prepare(for:sender:)` method as follows:

```
if let moviesVC = segue.destination as? MoviesViewController,  
    let familyMember = fetchedResultsController?.object(at:  
selectedIndex) {
```

```
moviesVC.persistentContainer = persistentContainer
moviesVC.familyMember = familyMember
}
```

This makes sure that a valid family member is passed to the movies view controller. Update the table view data source methods as shown in the following code. A fetched-results controller can retrieve objects based on an index path. This makes it a great fit to use in combination with table views and collection views:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return fetchedResultsController?.fetchedObjects?.count ?? 0
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    guard let cell = tableView.dequeueReusableCell(withIdentifier: "FamilyMemberCell"),
          let familyMember = fetchedResultsController?.object(at: indexPath)
    else { fatalError("Wrong cell identifier requested") }

    cell.textLabel?.text = familyMember.name

    return cell
}
```

If you run your app now, the interface updates automatically when you add a new family member to the database. However, the list of favorite movies doesn't update yet. That page does not use a fetched-results controller so it must listen to changes to the dataset directly.

The reason `MoviesViewController` doesn't use a fetched-results controller for the movie list is that fetched-result controllers will always need to drop down all the way to your persistent store (SQLite in this app). As mentioned before, querying the database has a significant memory overhead compared to traversing the relationship between family members and their movies; it's much faster to read the `movies` property than fetching them from the database.

Whenever a managed object context changes, a notification is posted to the default `NotificationCenter`. `NotificationCenter` is used to send events inside of an app so other parts of the code can react to those events.



It can be very tempting to use notifications instead of delegates, especially if you're coming from a background that makes heavy use of events such as JavaScript. Don't do this; delegation is better-suited to most cases, and it will make your code much more maintainable. Only use notifications if you don't care who's listening to your notifications or if setting up a delegate relationship between objects would mean you'd create very complex relationships between unrelated objects just to set up the delegation.

Let's subscribe `MoviesViewController` to changes in the managed object context so it can respond to data changes if needed. Before you implement this, add the following method, which should be called when changes in the managed object context occur:

```
extension MoviesViewController {
    @objc func managedObjectContextDidChange(notification: NSNotification) {
        guard let userInfo = notification.userInfo,
              let updatedObjects = userInfo[NSUpdatedObjectsKey] as? Set<FamilyMember>,
              let familyMember = self.familyMember
        else { return }

        if updatedObjects.contains(familyMember) {
            tableView.reloadData()
        }
    }
}
```

This method reads the notification's `userInfo` dictionary to access the information that's relevant to the current list. You're interested in changes to the current `familyMember` because when this object changes, you can be pretty sure that a new movie was just inserted. The `userInfo` dictionary contains keys for the inserted, deleted, and updated objects. In this case, you should look for the updated objects because users can't delete or insert new family members in this view. If the family member was updated, the table view is reloaded so it shows the new data.

The following code subscribes `MoviesViewController` to changes in the persistent container's managed object context:

```
override func viewDidLoad() {
    super.viewDidLoad()

    NotificationCenter.default.addObserver(self, selector:
#selector(self.managedObjectContextDidChange(notification:)), name:
```

```
.NSManagedObjectContextObjectsDidChange, object: nil)
}
```

When the view loads, the current `MoviesViewController` instance is added as an observer to the `.NSManagedObjectContextObjectsDidChange` notification. Go ahead and build your app; you should now see the user interface update whenever you add new data to your database.

Understanding the use of multiple instances of `NSManagedObjectContext`

It has been mentioned several times in this chapter that you can use multiple managed object contexts. In many cases, you will only need a single managed object context. Using a single managed object context means that all of the code related to the managed object context is executed on the main thread. If you're performing small operations, that's fine. However, imagine importing large amounts of data. An operation such as that could take a while. Executing code that runs for a while on the main thread will cause the user interface to become unresponsive. This is not good, because the user will think your app has crashed. So how do you work around this? The answer is using multiple managed object contexts.

In the past, using several managed object contexts was not easy to manage; you had to create instances of `NSManagedObjectContext` using the correct queues yourself. Luckily, `NSPersistentContainer` helps to make complex setups a lot more manageable. If you want to import data on a background task, you can obtain a managed object context by calling `newBackgroundContext()` on the persistent container. Alternatively, you can call `performBackgroundTask` on the persistent container and pass it a closure with the processing you want to do in the background.

One important thing to understand about Core Data, background tasks, and multithreading is that you must always use a managed object context on the same thread it was created on. Consider the following example:

```
let backgroundQueue = DispatchQueue(label: "backgroundQueue")
let backgroundContext = persistentContainer.newBackgroundContext()
backgroundQueue.async {
    let results = try? backgroundContext.fetch(someRequest)

    for result in results {
        // use result
    }
}
```

```
    }  
}
```

The behavior of this code can cause you a couple of headaches. The background context was created on a different queue than the one it's used on. It's always best to make sure to use a managed object context on the same queue it was created on by using the `perform(_:)` method of `NSManagedObject`. More importantly, you must also make sure to use the managed objects you retrieve on the same queue that the managed object context belongs to.

Often, you'll find that it's best to fetch data on the main queue using the `viewContext` persistent containers. Storing data can be delegated to background contexts if needed. If you do this, you must make sure that the background context is a child context of the main context. When this relationship is defined between the two contexts, your main context will automatically receive updates when the background context is persisted. This is quite convenient because it removes a lot of manual maintenance, which keeps your contexts in sync. Luckily, the persistent container takes care of this for you.

When you find that your app requires a setup with multiple managed object contexts, it's essential to keep the rules mentioned in this section in mind. Bugs related to using managed objects or managed object contexts in the wrong places are often tedious to debug and hard to discover. When implemented carefully, complex setups with multiple managed object contexts can increase your application's performance and flexibility.

Summary

This chapter has shown you how to implement a relatively simple Core Data database that stores family members and their favorite movies. You used the Core Data model editor in Xcode to configure the models you wanted to store and define the relationship between these models. Once the models were set up, you implemented code that created instances of your models so they could be stored in the database and retrieved later.

Next, you retrieved the data from the database and saw that your table view's don't automatically update when the underlying data changes. You used an `NSEntityDescription` controller to fetch family members and listen to changes on the list of family members. You saw that this setup is very powerful because you can respond to changes in your data quite easily.

In the next chapter, you will learn how to enrich the data your users add to the database by fetching and storing data from the web.

Questions

1. What does a persistent store coordinator do?
 - a) It talks to the SQLite database.
 - b) It talks to the persistent store.
 - c) It manages the data model file.
2. What is the function of the managed object context?
 - a) It shows instances of managed objects on the screen.
 - b) It mediates between managed objects and the persistent store coordinator.
 - c) It responds to changes in the persistent store.
3. How many managed object contexts can you use in an app?
 - a) Three.
 - b) It depends on the number of CPU cores in the device.
 - c) An unlimited amount.
4. What is the correct way to call `save()` on a managed object context?
 - a) `try! save()`.
 - b) `do save()`.
 - c) `try save()`.
5. When should you use a fetched-results controller?
 - a) Any time you want to show data on the screen and react to changes.
 - b) Only when you need to fetch data from the database and want to react to changes.
 - c) Any time you want to fetch data.
6. Does a fetched-results controller always go to the database to fetch data?
 - a) No, Core Data implements clever caching.
 - b) No, only when there is new data.
 - c) Yes, all fetch requests must go to the database.

Further reading

- The **Core Data** book, by Florian Kugler and Daniel Eggert
- Apple's Predicate Programming Guide: <http://apple.co/2fF3qHc>

13

Fetching and Displaying Data from the Network

Most modern applications communicate with a web service. Some apps rely on them heavily, acting as a layer that merely reads data from the web and displays it in app form. Other apps use the web to retrieve and sync data to make it locally available, and others only use the web as backup storage. Of course, there are a lot more reasons to use data from the internet than the ones mentioned.

In this chapter, you will expand the **MustC** application, so it uses a web service to retrieve popularity ratings for the movies that family members have added as their favorites. These popularity ratings will be stored in the Core Data database and displayed together with the names of the movies.

In this chapter, you'll learn about the following topics:

- URLSession
- Working with JSON in Swift
- Updating Core Data objects with fetched data

Fetching data from the web

Retrieving data from the web is something that you will often do as an iOS professional. You won't just fetch data from a web service; you'll also send data back to it. For example, you might have to make an HTTP POST request as part of a login flow or to update a user's profile information. Over time, iOS has evolved quite a bit in the web requests department, making it easier to use web services in apps.



HTTP (or HTTPS) is a protocol that almost all web traffic uses for communication between a client, such as an app, and a server. The HTTP protocol supports several methods that signal the request's intent. GET is used to retrieve information from a server. A POST request indicates the intention to push new content to a server, for instance, submitting a form.

When you want to perform a web request in iOS, you will typically use the URLSession class. The URLSession class makes asynchronous web requests on your behalf. This means that iOS loads data from the web on a background thread, ensuring that the user interface remains responsive throughout the entire request. If a web request is performed synchronously, the user interface is unresponsive for the duration of the network request because a thread can only do one thing at a time, so if it's waiting for a response from the network, it can't respond to touches or other user input.

If your user has a slow internet connection, a request could take several seconds. You don't want your interface to freeze for several seconds. Even a couple of milliseconds will create a noticeable drop in its responsiveness and frame rate. This can be easily avoided with URLSession to perform asynchronous network requests.

First, you will experiment with basic network requests in a playground. You can create a new playground or use the one provided in this book's code bundle. After you've seen the basics of URLSession, you'll implement a way to fetch movies from an open source movie database and put this implementation to use in the MustC app.

Understanding URLSession basics

The following code snippet shows a sample of a network request that loads the <https://apple.com> homepage:

```
import Foundation

let url = URL(string: "https://apple.com")!
let task = URLSession.shared.dataTask(with: url)
{
    data, response, error in
    print(data)
    print(response)
    print(error)
}

task.resume()
```

This is an elementary example: A URL is created, and then the shared `URLSession` instance is used to create a new `dataTask`. This `dataTask` is an instance of `URLSessionDataTask` and allows you to load data from a remote server.

Alternatively, you could use a download task if you're downloading a file, or an upload task if you're uploading files to a web server. After creating the task, you must call `resume` on the task, because new tasks are always created in a suspended state.

If you run this sample in an empty playground, you'll find that the example doesn't work. Because the network request is made asynchronously, the playground finishes its execution before the network request is complete. To fix this, you should make sure that the playground runs indefinitely. Doing so will allow the network request to finish. Add the following lines to the top of the playground source file to enable this behavior:

```
import PlaygroundSupport  
PlaygroundPage.current.needsIndefiniteExecution = true
```

Now that the playground runs indefinitely, you'll find that there isn't a lot of useful data printed to the console. In this case, you're not interested in the raw data, HTTP headers, or the fact that the error is `nil`. When you load data from a URL, you're often most interested in the response's body. The body of a response usually contains the string representation of the data you requested. In the case of the preceding example, the body is the HTML that makes up Apple's homepage. Let's see how you can extract this HTML from the response. Replace the data task's completion callback with the following:

```
{ data, response, error in  
    guard let data = data, error == nil  
    else { return }  
  
    let responseString = String(data: data, encoding: .utf8)  
    print(responseString)  
}
```

The preceding callback closure makes sure that there are no errors returned by the web service and that there is data present. Then, the raw data is converted to a string, and that string is printed to the console. If you use this callback instead of the old one, you'll see the HTML for the Apple homepage printed. Simple requests to a web server like the one you just saw are relatively simple to implement with `URLSession`.

You can take more control over the request that's executed by creating a `URLRequest` instance. The example you saw is one where you let `URLSession` create the `URLRequest` on your behalf. This is fine if you want to perform a simple HTTP GET request with no custom headers, but if you're going to post data or include specific headers, you will need to have more control over the request that's used. Let's take a look at what a GET request with some parameters and a custom header looks like.

The following code uses an API key from <https://www.themoviedb.org/>. If you want to try this code example, create an account on their website and request an API key on your account page. Setting this up should only take a couple of minutes, and if you want to follow along with this chapter, you will need to have your own API key:

```
let api_key = "YOUR_API_KEY_HERE"
var urlString = "https://api.themoviedb.org/3/search/movie/"
urlString = urlString.appending("?api_key=\(api_key)")
urlString = urlString.appending("&query=Swift")

let movieURL = URL(string: urlString)!

var urlRequest = URLRequest(url: movieURL)
urlRequest.httpMethod = "GET"
urlRequest.setValue("application/json", forHTTPHeaderField: "Accept")

let movieTask = URLSession.shared.dataTask(with: urlRequest) { data,
    response, error in
    print(response)
}

movieTask.resume()
```

The preceding code is a bit more complicated than the example you saw before. In this example, a more complex URL is configured that includes some HTTP GET parameters. The `httpMethod` for `URLRequest` is specified, and a custom header is provided to inform the receiver of this request about the type of response it would like to receive.

The flow for executing this URL request is the same as the one you saw earlier. However, the URL that is loaded responds with a JSON string instead of an HTML document. JSON is used by many APIs as the preferred format to pass data around on the web. In order to use this response, the raw data must be converted to a useful data structure. In this case, a dictionary will do. If you haven't seen or worked with JSON before, it's a good idea to take a step back and read up on the JSON data format because this chapter will continue under the assumption that you are at least somewhat familiar with JSON.

Working with JSON in Swift

The following snippet shows how you can convert raw data to a JSON dictionary. Working with JSON in Swift can be a little tedious at times, but overall, it's an alright experience. Let's look at an example:

```
guard let data = data,
      let json = try? JSONSerialization.jsonObject(with: data, options:
[])
else { return }

print(json)
```

The preceding snippet converts the raw data that is returned by a URL request to a JSON object. The print statement prints a readable version of the response data, but it's not quite ready to be used. Let's see how you gain access to the first available movie in the response.

If you look at the type of object returned by the `jsonObject (with:options:)` method, you'll see that it returns `Any`. This means that you must typecast the returned object to something you can work with, such as an array or a dictionary. When you inspect the JSON response that the API returned, for instance by using `print` to make it appear in the console like you did with Apple's homepage HTML, you'll notice that there's a dictionary that has a key called `results`. The `results` object is an array of movies. In other words, it's an array of `[String: Any]`, because every movie is a dictionary, where strings are the keys and the value can be a couple of different things, such as `Strings`, `Int`, or `Booleans`. With this information, you can access the first movie's title in the JSON response, as shown in the following code:

```
guard let data = data,
      let json = try? JSONSerialization.jsonObject(with: data, options:
[]),
      let jsonDict = json as? [String: AnyObject],
      let resultsArray = jsonDict["results"] as? [[String: Any]]
else { return }

let firstMovie = resultsArray[0]
let movieTitle = firstMovie["title"] as! String
print(movieTitle)
```

Working with dictionaries to handle JSON isn't the best experience. Since the JSON object is of the `AnyObject` type and you need to typecast every element in the dictionary you want to access, there's a lot of boilerplate code you need to add. Luckily, Swift has better ways to create instances of objects from the JSON data. The following example shows how you can quickly create an instance of a `Movie` struct without having to cast all the keys in the JSON dictionary to the correct types for the `Movie` struct.

First, let's define two structs, one for the `Movie` itself, and one for the response that contains the array of `Movie` instances:

```
struct MoviesResponse: Codable {  
    let results: [Movie]  
}  
  
struct Movie: Codable {  
    let id: Int  
    let title: String  
    let popularity: Float  
}
```

Next, you can use the following snippet to quickly convert the raw data from a URL request to an instance of `MoviesResponse`, where all movies are converted to instances of the `Movie` struct:

```
let decoder = JSONDecoder()  
guard let data = data,  
    let movies = try? decoder.decode(MoviesResponse.self, from: data)  
else { return }  
  
print(movies.results[0].title)
```

You might notice that both `MoviesResponse` and `Movie` conform to the `Codable` protocol. The `Codable` protocol was introduced in Swift 4, and it allows you to easily encode and decode data objects. The only requirement is that all properties of a `Codable` object conform to the `Codable` protocol. A lot of built-in types, such as `Array`, `String`, `Int`, `Float`, and `Dictionary` conform to `Codable`. Because of this, you can easily convert an encoded JSON object into a `MoviesResponse` instance that holds `Movie` instances.

By default, each property name should correspond to the key of the JSON response it is mapped to. However, sometimes you might want to customize this mapping. For instance, the `poster_path` property in the response we've been working with so far would be best mapped to a `posterPath` property on the `Movie` struct. The following example shows how you would tackle these circumstances:

```
struct Movie: Codable {  
  
    enum CodingKeys: String, CodingKey {  
        case id, title, popularity  
        case posterPath = "poster_path"  
    }  
  
    let id: Int  
    let title: String  
    let popularity: Float  
    let posterPath: String?  
}
```

By specifying a `CodingKeys` enum, you can override how the keys in the JSON response should be mapped to your `Codable` object. You must cover all keys that are mapped, including the ones you don't want to change. As you've seen, the `Codable` protocol provides powerful tools for working with data from the network. Custom key mapping makes this protocol even more powerful because it allows you to shape your objects exactly how you want them instead of having the URL responses dictate the structure to you.

If the only conversion you need to apply in the coding keys is converting from snake case (`poster_path`) to camel case (`posterPath`), you don't have to specify the coding keys yourself. The `JSONEncoder` object can automatically apply this type of conversion when decoding data if you set its `keyDecodingStrategy` to `.convertFromSnakeCase`, as shown in the following code:

```
let decoder = JSONDecoder()  
decoder.keyDecodingStrategy = .convertFromSnakeCase
```

Try implementing this in your playground and remove `CodingKeys` from the `Movie` object to ensure your JSON decoding still works.

Now let's move on to storing fetched data in the Core Data database.

Updating Core Data objects with fetched data

So far, the only thing you have stored in Core Data is movie names. You will expand this functionality by performing a lookup for a certain movie name through the movie database API. The fetched information will be used to display and store a popularity rating for the movies in the Core Data database.

A task such as this seems straightforward at first; you could come up with a flow such as the one shown in the following steps:

1. A user indicates their favorite movie.
2. The movie's popularity rating is fetched.
3. The movie and its rating are stored in the database.
4. The interface updates with the new movie.

At first glance, this is a fine strategy; insert the data when you have it. However, it's important to consider that API calls are typically done asynchronously so the user interface stays responsive. More importantly, API calls can be really slow if your user doesn't have a good internet connection. This means that you would be updating the interface with very noticeable lag if the preceding steps are executed one by one.

The following would be a much better approach to implement the feature at hand:

1. The user indicates their favorite movie.
2. The user stores the movie.
3. Update the interface with the new movie.
4. Begin popularity fetching.
5. Update the movie in the database.
6. Update the interface with the popularity.

This approach is somewhat more complex, but it will give the user a responsive experience. The interface will respond to new movies immediately by showing them, and then automatically update as soon as new data is retrieved. Before you can fetch the data and update the models, the Core Data model must be updated in order to store the movie's popularity rating.

Open the Core Data model editor and select the `Movie` entity. All you have to do is add a new property and name it **popularity**. Select the `Double` type for this property because the **popularity** is stored as a decimal value. You have to make sure that this property is optional since you won't be able to provide a value for it straight away:

The screenshot shows the Xcode Core Data model editor. On the left, there's a sidebar with sections for ENTITIES, FETCH REQUESTS, and CONFIGURATIONS. Under ENTITIES, 'FamilyMember' and 'Movie' are listed; 'Movie' is selected and highlighted with a grey background. In the main area, under the 'Attributes' section, there are two entries:

Attribute	Type
N popularity	Double
S title	String

Below the table are '+' and '-' buttons for adding or removing attributes.

If you've worked with Core Data prior to when iOS 10 was released, this is the part where you expect to read about migrations and how you can orchestrate them. However, for simple changes such as this, we don't need to manage migrations. All you need to do is simply build and run your application to regenerate your model definitions, and for a simple change, such as the one we performed just now, Core Data will automatically manage the migration on its own.



If you want to support iOS versions earlier than 10, make sure you read up on Core Data migrations. Whenever you update your models, you have to make sure that your database can properly migrate from one model version to another. During development, this isn't extremely important: you just reinstall the app whenever your models change. However, app updates will crash on launch if the Core Data model isn't compatible with the previous model.

Now that the model is updated, let's figure out how to implement the flow that was described earlier.

Implementing the fetch logic

The asynchronous nature of network requests makes certain tasks, such as the one you're about to implement, quite complex. Usually, when you write code, its execution is very predictable. Your app typically runs line by line, so any line that comes after the previous one can assume that the line before it has finished executing. This isn't the case with asynchronous code. Asynchronous code is taken off the main thread and runs separately from the rest of your code. This means that your asynchronous code might run in parallel with other code. In the case of a network request, the asynchronous code might execute seconds after the function that initiated the request.

This means that you need to figure out a way to update and save movies that were added as soon as the rating has been retrieved. What's interesting about this is that once you see the code that implements this feature, it will feel natural to you that this is how it works. However, it's important that you're aware of the fact that it's not as straightforward as it may seem at first.

It's also important that you're aware of the fact that the code you're about to look at is executed on multiple threads. This means that even though all pieces of the code are defined in the same place, they are not executed on the same thread. The callback for the network request is executed on a different thread than the code that initiated the network request. You have already learned that Core Data is not thread-safe. This means that you can't safely access a Core Data object on a different thread than the thread it was created on.

If this confuses you, that's okay. You're supposed to be a bit confused right now. Asynchronous programming is not easy, and fooling you into thinking it is will cause frustration once you run into concurrency-related troubles (and you will). Whenever you work with callbacks, closures, and multiple threads, you should be aware that you're doing complex work that isn't straightforward.

Now that you understand that asynchronous code is hard, let's take a closer look at the feature you're about to implement. It's time to start implementing the network request that fetches popularity ratings for movies. You will abstract the fetching logic into a helper named `MovieDBHelper`. Go ahead and create a new helper folder in Xcode and add a new Swift file called `MovieDBHelper.swift` to it.

Abstracting this logic into a helper has multiple advantages. One of them is simplicity; it will keep our view controller code nice and clean. Another advantage is flexibility. Let's say that you want to combine multiple rating websites, or a different API, or compute popularity based on the number of family members who added this same title to their list; it will be easier to implement since all the logic for ratings is in a single place.

Add the following skeleton implementation to the MovieDBHelper file:

```
struct MovieDBHelper {
    typealias MovieDBCallback = (Double?) -> Void
    let apiKey = "YOUR_API_KEY_HERE"

    func fetchRating(forMovie movie: String, callback: @escaping
        MovieDBCallback) {

    }

    private func url(forMovie movie: String) -> URL? {
        guard let query =
            movie.addingPercentEncoding(withAllowedCharacters: .urlHostAllowed)
            else { return nil }

        var urlString = "https://api.themoviedb.org/3/search/movie/"
        urlString = urlString.appending("?api_key=\(apiKey)")
        urlString = urlString.appending("&query=\(query)")

        return URL(string: urlString)
    }
}
```

The preceding code starts off with an interesting line:

```
typealias MovieDBCallback = (Double?) -> Void
```

This line specifies the type that's used for the callback closure that's called when the rating is fetched. This callback will receive an optional Double as its argument. If the network request fails for any reason, the Double will be nil. Otherwise, it contains the rating for the movie that the request was created for.

The snippet also contains a dummy method that performs the fetch; you will implement this method soon. Finally, there's a method that builds a URL. This method is private because it's only supposed to be used inside of the helper struct. Note that the movie is converted to a percent-encoded string. This is required because if your user adds a movie with spaces in it, you would end up with an invalid URL if the spaces aren't properly encoded.

Before you implement `fetchRating(forMovie:callback)`, add a new file named `MovieDBResponse.swift` to the helper folder. This file will be used to define a struct that represents the response we expect to receive from the Moviedb API. Add the following implementation to this file:

```
struct MovieDBLookupResponse: Codable {  
  
    struct MovieDBMovie: Codable {  
        let popularity: Double?  
    }  
  
    let results: [MovieDBMovie]  
}
```

The preceding code uses a nested struct to represent the movie objects that are part of the response. This is similar to what you saw in the playground example. Structuring the response this way makes the intent of this helper very obvious, which usually makes code easier to reason about. With this struct in place, let's see what the implementation of `fetchRating(forMovie:callback)` looks like in the following code:

```
func fetchRating(forMovie movie: String, callback: @escaping  
MovieDBCallback) {  
    guard let searchUrl = url(forMovie: movie) else {  
        callback(nil)  
        return  
    }  
  
    let task = URLSession.shared.dataTask(with: searchUrl) { data,  
response, error in  
        var rating: Double? = nil  
  
        defer {  
            callback(rating)  
        }  
  
        let decoder = JSONDecoder()  
  
        guard error == nil, let data = data,  
            let lookupResponse = try?  
decoder.decode(MovieDBLookupResponse.self, from: data),  
            let popularity = lookupResponse.results.first?.popularity  
            else { return }  
  
        rating = popularity  
    }  
}
```

```
    task.resume()  
}
```

This implementation looks very similar to what you experimented with earlier in the playground. The URL-building method is used to create a valid URL. If this fails, it makes no sense to attempt requesting the movie's rating, so the callback is called with a `nil` argument. This will inform the caller of this method that the execution is done and no result was retrieved.

Next, a new data task is created and `resume` is called on this task to kick it off. There is an interesting aspect to how the callback for this data task is called, though. Let's take a look at the following lines of code:

```
var rating: Double? = nil  
  
defer {  
    callback(rating)  
}
```

A `rating` double is created here, and it is given an initial value of `nil`. Then there's a `defer` block. The code inside of the `defer` block is called right before exiting the scope. In other words, it's executed right before the code returns from a function or closure.

Since this `defer` block is defined inside the callback for the data task, the callback for the `fetchRating(forMovie:callback:)` method is always called just before the data task callback is exited. This is convenient because all you must do is set the value for the rating to a double, and you don't have to manually invoke the callback for each possible way the scope can be exited. This also applies when you return because of unmet requirements. For instance, if there is an error while calling the API, you don't need to invoke the callback. You can simply return from the closure, and the callback is called automatically. This strategy can also be applied if you instantiate or configure objects temporarily and you want to perform some clean-up when the method, function, or closure is done.

The rest of the code should be fairly straightforward since most of it is nearly identical to the code used in the playground. Now that you have the networking logic down, let's take a look at how to actually update the movie object with a popularity rating.

Updating a movie with a popularity rating

To update the movie object, you will implement the final step of the approach that was outlined earlier. You need to asynchronously fetch a rating from the movie database and then use that rating to update the movie. The following code should be added to `MoviesViewController.swift`, right after adding a new movie to a family member, inside of the `persist` block, approximately at line 35 (depending on how you have formatted your code):

```
let helper = MovieDBHelper()
helper.fetchRating(forMovie: name) { rating in
    guard let rating = rating
    else { return }

    moc.persist {
        movie.popularity = rating
    }
}
```

You can see that the helper abstraction provides a nice interface for the view controller. You can simply use the helper and provide it a movie to fetch the rating for with a callback and you're all set. Abstracting code like this can make maintaining your code a lot more fun in the long run.

The most surprising thing in the preceding snippet is that `moc.persist` is called again inside of the helper callback. This must be done because this callback is actually executed long after the initial persist has finished. Actually, this callback isn't even executed on the same thread as the code it's surrounded by.

To see how your code fails if you don't properly persist your model, try replacing the `moc.persist` block in the rating retrieval callback with the following code:

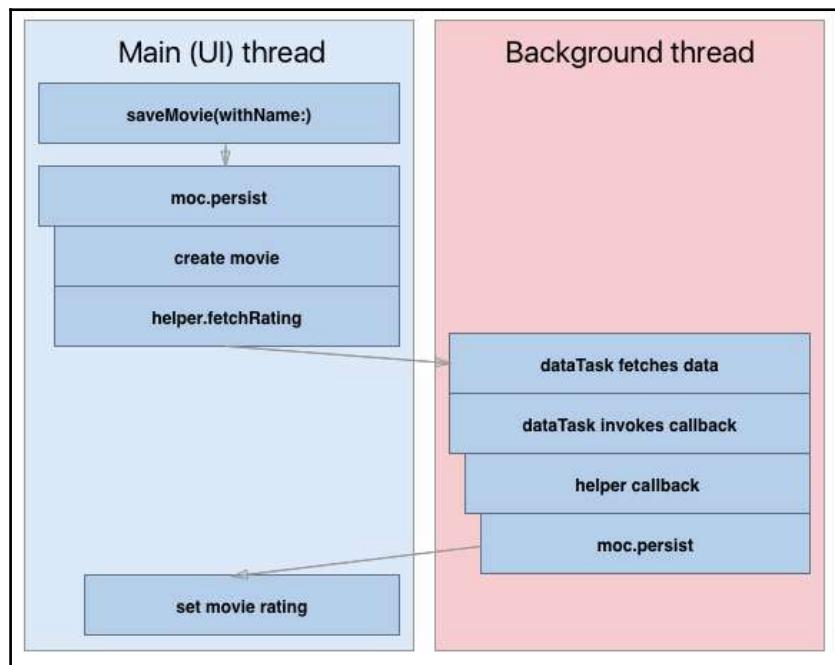
```
movie.popularity = rating
do {
    try moc.save()
} catch {
    moc.rollback()
}
```

If you add a new movie now, the rating will still be fetched. However, you will suddenly run into issues when reloading your table view. This is because the managed object context was saved on a background thread. This means that the notification that informs the table view about updates is also sent on a background thread. You could resolve the issue by pushing the `reloadData()` call onto the main thread as you've done before, but in this case, doing so would only make the problem worse. Your app might work fine for a while, but once your app grows in complexity, using the same managed object context in multiple threads will most certainly cause crashes. Therefore, it's important to always make sure that you access managed objects and their contexts on the correct thread by using a construct, such as the `persist` method we implemented for this app.

Now that you have looked at all the code involved, let's see what all this threading talk means in a more visual way.

Visualizing multiple threads

The following diagram will help you understand multiple threads:



When `saveMovie(withName:)` is called, the execution is still on the main thread. The persistence block is opened, the movie is created, its name is set, a helper is created, and then `fetchRating(forMovie:callback:)` is called on the helper. This call itself is still on the main thread. However, the fetching of data is pushed to a background thread. This was discussed earlier when you experimented with fetching data in a playground.

The callback that's invoked by `dataTask` is called on the same background thread that the task itself is on. The code will do its thing with the JSON and finally, the callback that was passed to `fetchRating(forMovie:callback:)` is called. The code inside of this callback is executed on the background thread as well.

You can see that the set movie-rating step in the update flow is somehow pushed back to the main thread. This is because of the `persist` method that you added as an extension to the managed object context. The context uses the `perform` method internally to ensure that any code we execute inside of the `persist` block is executed on the thread the managed object context is on. Also, since the managed object context was created on the main thread, the movie rating will be set on the main thread.



If you didn't set the movie rating on the same thread that the managed object belongs to, you would get errors and undefined behavior. Always make sure that you manipulate Core Data objects on the same thread as their managed object context.

Threading is a complex subject, but it's essential for building responsive applications. Network logic is a great example of why multithreading is important. If we didn't perform the networking on a separate thread, the interface would be unresponsive for the duration of the request. If you have other operations that might take a while in your app, consider moving them onto a background thread so they don't block the user interface.

Wrapping the feature up

All of the code is in place, and you have a better understanding of multithreading and how callbacks can be used in a multithreaded environment. Yet, if you build and run your app and add a new movie, the rating won't be displayed yet.

The following are the three reasons why this is happening:

- The table view cell that shows the movie isn't updated yet.
- The network request doesn't succeed because of App Transport Security.
- Updates to movie objects aren't observed yet.

Let's solve these issues in order, starting with the table view cell.

Adding the rating to the movie cell

Currently, the movie table view displays cells that have a title. `UITableViewCell` has a built-in option to display a title and a subtitle for a cell.

Open `Main.storyboard` and select the prototype cell for the movies. In the `Attributes Inspector` field, change the cell's style from basic to subtitle. This will allow you to use `detailTextLabel` on the table view cell. This is where we'll display the movie rating.

In `MoviesViewController`, add the following line to `tableView(_:cellForRowAt:)`, right after you set the cell's title:

```
cell.detailTextLabel?.text = "Rating: \(movie.popularity)"
```

This line will put the movie's popularity rating in a string and assign it as a text for the detail text label.

If you build and run your app now, all movies should have a popularity of 0.0. Let's fix this by resolving the networking issue.

Understanding App Transport Security

With iOS 9, Apple introduced **App Transport Security (ATS)**. ATS makes applications safer and more secure by prohibiting the use of non-HTTPS resources. This is a great security feature, as it protects your users from a wide range of attacks that can be executed on regular HTTP connections.

If you paid close attention to the URL that's used to fetch movies, you may have noticed that the URL should be an HTTPS resource, so it should be fine to load this URL. However, the network requests are still blocked by ATS. Why is this?

Well, Apple has strict requirements. At the time of writing this book, the movie database uses the SHA-1 signing of certificates, whereas Apple requires SHA-2. Because of this, you will need to circumvent ATS for now. Your users should be safe regardless, since the movie database supports HTTPS, just not the version Apple considers to be secure enough.

To do this, open the `Info.plist` file and add a new dictionary key named **App Transport Security Settings**. In this dictionary, you will need an **Exception Domains** dictionary. Add a new dictionary key named **themoviedb.orgr** to this dictionary and add two Booleans to this dictionary. Both should have **YES** as their values, and they should be named **NSIncludesSubdomains** and **NSTemporaryExceptionAllowsInsecureHTTPLoads**. Refer to the following screenshot to make sure that you've set this up correctly:

▼ App Transport Security Settings	Dictionary	(1 item)
▼ Exception Domains	Dictionary	(1 item)
▼ themoviedb.org	Dictionary	(2 items)
NSIncludesSubdomains	Boolean	YES
NSTemporaryExceptionAllow...	Boolean	YES

If you add a new movie to a family member now, nothing updates yet. However, if you go back to the family overview and then back to the family member, you'll see that the rating for the most recent movie is updated. Great! Now, all you need to do is make sure that we observe the managed object context for updates to the movies so they are reloaded if their rating changes.

Observing changes to movie ratings

You're already observing the managed object context for changes, but they are only processed if the family member that is shown on the current page has updated. This logic should be replaced so that it will reload the table view if either the family member or their favorite movies change. Update the `managedObjectContextDidChange(_:)` method in `MoviesViewController.swift` as follows:

```
@objc func managedObjectContextDidChange(notification: NSNotification)
{
    guard let userInfo = notification.userInfo
        else { return }

    if let updatedObjects = userInfo[NSUpdatedObjectsKey] as?
        Set<FamilyMember>,
```

```
let familyMember = self.familyMember,  
updatedObjects.contains(familyMember) {  
  
    tableView.reloadData()  
}  
  
if let updatedObjects = userInfo[NSUpdatedObjectsKey] as? Set<Movie>  
{  
    for object in updatedObjects {  
        if object.familyMember == familyMember {  
            tableView.reloadData()  
            break  
        }  
    }  
}
```



The logic for observing the family member hasn't changed; its conditions simply moved from the guard statement to an `if` statement. An extra `if` statement was added for the movies. If the updated object set is a list of movies, we loop through the movies and check whether one of the movies has the current family member as its family member. If so, the table is refreshed immediately and the loop is exited.

It's important that the loop in the second `if` statement is set up like this because you might have just added a movie for family member A and then switched to family member B while the new movie for family member A is still loading its rating. Also, breaking out of the loop early ensures that you don't loop over any more objects than needed. All you want to do is refresh the table view if one of the current family members' favorite movies is updated.

Okay, build and run your app to take it for a spin! You'll notice that everything works as you'd want it to right now. Adding new movies triggers a network request; as soon as it finishes, the UI is updated with the new rating. Sometimes, this update will be done in an instant, but it could take a short while if you have a slow internet connection. Great! That's it for this feature.

Summary

This chapter was all about adding a small, simple feature to an existing app. We added the ability to load real data from an API. You saw that networking is made pretty straightforward by Apple with `URLSession` and data tasks. You also learned that this class abstracts away some very complex behavior regarding multithreading, so your apps remain responsive while data is loaded from the network. Next, you implemented a helper struct for networking and updated the Core Data model to store ratings for movies. Once all this was done, you could finally see how multithreading worked in the context of this app. This wasn't everything we needed to do, though. You learned about ATS and how it keeps your users secure. You also learned that you sometimes need to circumvent ATS, and we covered how you can achieve this.

Even though the feature itself wasn't very complex, the concepts and theory involved can be quite overwhelming. You suddenly had to deal with code that would be executed asynchronously in the future. And not just that. The code even used multiple threads to make sure that its performance was optimal. The concepts of multithreading and asynchronous programming are arguably two of the more complex aspects of programming. Practice them a lot and try to remember that any time you're passing around a closure, you could be writing some asynchronous code that gets executed on a different thread.

Now that the list of movies is updated with data from the web, let's take it one step further in the next chapter. You can use background-fetching to make sure that you always fetch the most up-to-date information for movies and update them without the user even noticing.

Questions

1. What object should you use to make web requests?
 - a) `URLSession`.
 - b) `URLRequest`.
 - c) `WebTask`.
2. How are web requests executed?
 - a) Asynchronously.
 - b) Synchronously.

3. What is a typealias?
 - a) A closure callback.
 - b) A tuple.
 - c) An alias for any other type, for instance a callback.
4. What happens if you use managed objects on the wrong thread?
 - a) Nothing.
 - b) They won't save.
 - c) Your app might crash.
5. Which object do you need if you make an HTTP POST request?
 - a) POSTURLRequest.
 - b) URLRequest.
 - c) POSTRequest.
6. What is ATS?
 - a) An encryption protocol.
 - b) Apple's way to enforce secure web requests.
 - c) Apple's name for HTTPS.

Further reading

- Chapter 28, Offloading Tasks with Operations and GCD

14

Being Proactive with Background Fetch

So far, the **MustC** application is shaping up quite nicely. You can add family members and associate movies with them, and these movies are automatically enriched with the first rating available from the movie database API. You may still have some movies in your app that were added earlier and don't have a rating, although the ratings for the existing movies might be correct for now, they might not be correct in a couple of days, weeks, or months from now. You could update the ratings whenever the user accesses them, but it's not very efficient to do this. You might reload the same movie a couple of times in a single session, which isn't very useful.

Ideally, the movies would be updated in the background, when the user isn't using the app. Functionality such as this can be implemented using background fetch. This chapter is all about fetching data on behalf of the user while the app isn't active in the foreground. Implementing this feature can significantly benefit your users because your app will have fresh content every time the user opens up the app. No pull to refresh is needed, and users love these little touches of magic in the apps they use.

This chapter covers the following topics:

- Understanding how background fetching works
- Implementing the prerequisites for background fetch
- Updating movies in the background

Besides implementing the background fetch feature, you will do a lot of refactoring. It's important to understand how and when to refactor your code, because it's highly unlikely that you'll be able to exactly nail a flexible setup for your app that can adapt to any requirement that is thrown at it. Being able to refactor your code whenever you see duplication or readability issues is an essential skill that you will utilize almost every day as your understanding of your code base, the requirements, and the iOS platform grows.

Understanding how background fetch works

Any application that provides users with some form of data that updates over time can implement background fetch. An application that implements background fetch is woken up by iOS periodically, and it's given a small window of time to fetch and process new data that has become available. The OS expects applications to call a callback when they're done with fetching and processing the data. The application uses this callback to inform the OS about whether or not it retrieved new data. First, you will learn about background fetch at a high level. Then, you'll learn about some of the details involved with background fetch, and lastly, you will implement background fetch in the **MustC** app.

Looking at background fetch from a distance

Background fetch allows apps to download new content in the background without draining the battery because iOS manages the wake-up intervals as efficiently as possible. Since iOS will not allow your application to stay active for a very long time, you must keep in mind that your app might not be able to perform all the tasks you would like it to perform. If this is the case, you're probably implementing background fetch for the wrong reasons. Sometimes, it's a better idea to split work between your background fetch and application launch.

One example of this would be to download a large amount of data and perform some expensive processing with the downloaded data. If your app takes too long to process the data, it won't be able to finish. It may be a good idea to continue processing once the app launches. An even better approach would be to download only the data and perform all of the expensive processing only when the app launches. Background fetch is intended to pull in quick, small updates and not to process large amounts of data.

The time intervals that iOS uses to wake your app up aren't entirely transparent. You can tell iOS how often you would like your app to be woken up to do some background work, but ultimately the OS itself decides whether or not your app is woken up and how much time it has to perform its tasks.

A neat feature of background fetch is that iOS learns how often your app has updated data because you have to report this to the completion callback. In addition to understanding the moments where your app is most likely to retrieve updates, iOS also learns when your users are most likely to use your app. If a user tends to open your app every morning around 10:00 A.M., chances are that your app will be woken up in the background sometime before 10:00 A.M. This feature makes sure that your users will see recent content for your application. Similarly, if iOS notices that your app rarely has new content within a specified time interval, it's likely that your app isn't woken up as often as you'd like.

Leaving this interval in the hands of the OS might seem a bit scary at first; you often want as much control over these things as possible. What if you know that once a month, at a particular time, you will publish a bunch of new content and you want your app to fetch this in the background? How will you know that all users will receive this new content in time? The answer to this is simple: you don't know. Background fetch is a service that attempts to help your users to receive content in the background. There are no guarantees.

You should always be aware that your app might have to update content in the foreground since users can turn off background fetching entirely. One more important aspect of background fetch is that it will only work if your app isn't completely closed. If a user opens the multitasking window and swipes your app upward to close it, it won't be woken up for a background refresh, and your app will have to refresh when the user launches it the next time.

An excellent way to facilitate this in your code is to implement your background fetch logic in a separate method or a helper that you can call. You could invoke a method that's responsible for fetching new content any time your application is launched, or when it's awakened to perform a background fetch. This strategy ensures that your users always receive the latest content when they open your app.

This is all you need to know about background fetching on the surface. It's a feature that periodically wakes up your application so it can perform a task for a short time. You don't have tight control over when and how your app is awakened, which is okay because iOS learns how often your app should be woken up, and more importantly, iOS learns the best time to wake your app up. All you have to do is make sure that iOS knows that your app wants to be woken up.

Looking at background fetch in more depth

Now that you have a broad understanding of background fetch; let's investigate what your app needs to do to be compatible with background fetch. First of all, you should let iOS know that your app wants to be woken up periodically. To do this, you will need to turn on the **Background Modes** capability for your app. Within this capability, you can opt in to background fetch. Enabling this mode will automatically add a key for this service to your app's `Info.plist`.

When this capability is enabled, iOS will allow your app to be woken up periodically. However, it won't magically work right away. In the `application(_:didFinishLaunchingWithOptions:)` method of `AppDelegate`, you must inform the system that you want the app to become active every once in a while and also specify the minimum time between fetches.

Once you've done this, iOS will periodically wake your app up. When it does, the `application(_:performFetchWithCompletionHandler:)` method is called. This method should be implemented in `AppDelegate`, and it's expected to call the provided completion handler once the work is completed. If you fail to call this method in time, iOS will make a note of this, and missing the window to call this completion-handler could result in your app not being woken up as often as you'd like, so make sure to always call it when the background work completes.

After calling the completion-handler, your app will be put back to sleep until the user opens the app or until the next time your app is allowed to perform a background fetch. Calling the completion-handler shouldn't be done until all of the work that you intended to do is done because once the completion-handler is called, there are no guarantees about how and when your app will make a transition to the background. The only thing you know for sure is that it will happen soon.

Implementing the prerequisites for background fetch

There are three steps involved in supporting background fetch in your app:

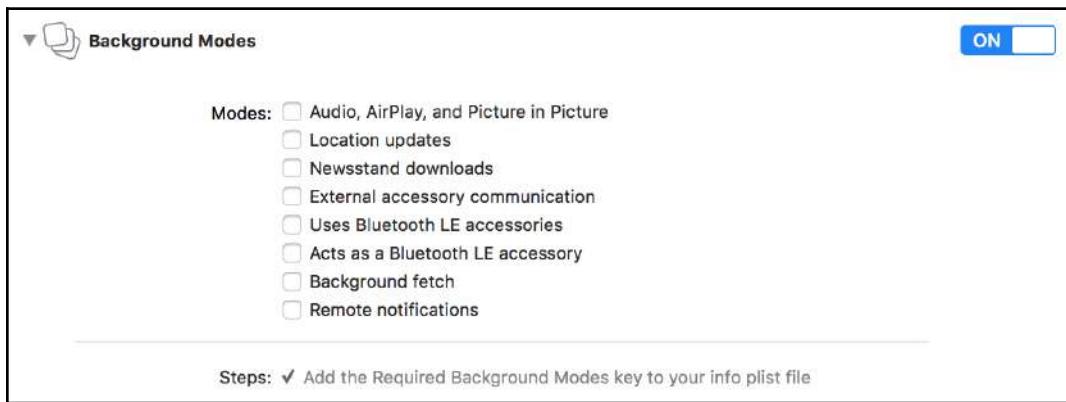
1. Add the background fetch capability to your app.
2. Ask iOS to wake your app up.
3. Implement `application(_:performFetchWithCompletionHandler:)` in `AppDelegate`.

You can implement step 1 and 2 right now; step 3 will be implemented separately because this step will involve writing the code to fetch and update the movies using a helper struct.

Adding the background fetch capability

Every application has a list of capabilities they can opt in to. Some examples of these capabilities are using Maps, HomeKit, and Background Modes. You can find the **Capabilities** tab in your project settings. If you select your project in the file navigator, you can see the **Capabilities** tab right next to your app's General settings.

If you select this tab, you will see a list of all the capabilities your app can implement. If you expand a capability, you'll find some information about what the capability does and what happens automatically if you enable it. If you expand the **Background Modes** capability, you'll see the following information:



There are several options that you can enable, but there's just one that you need to focus on for now. The capability you need to enable is **Background Fetch**. Click the on/off checkbox to enable the capability and then check the **Background Fetch** option.

Enabling **Background Fetch** isn't something that magically changes some invisible settings for your application. You can enable background fetch manually if you'd like to.

Open the `Info.plist` file and search for the **Required background modes** key. You can see that it contains a single item with a value of **App downloads content from the network**. This entry in the `Info.plist` file enables your app to ask iOS to wake it up periodically:

▼ Required background modes	Array	(1 item)
Item 0	String	App downloads content from the network

However, just because you can do this manually does not mean you should. The capabilities tab is a very convenient place to manage capabilities from and manually adding the right key to the `Info.plist` file is tedious and error-prone. It's interesting to know that there's no magic involved in this process, and it makes the entire feature feel more transparent.

Asking iOS to wake your app

You're halfway there in the process of enabling background fetching for **MustC**. All that's left is inform iOS that it should wake the app up every once in a while. This behavior should be implemented in the

`application(_: didFinishLaunchingWithOptions:)` method in `AppDelegate`. You must ask iOS to wake your app up in this method because you should ask iOS to wake your app up every time your app launches. When your app transitions from the foreground to the background normally, iOS will know that your app should wake up at certain times. However, if the app is killed entirely, iOS won't be able to wake the app up again, so you must ask iOS to wake your app up again as soon as the app launches again.

There isn't much code involved in enabling background fetching for your app. Add the following line to `application(_: didFinishLaunchingWithOptions:)`:

```
application.setMinimumBackgroundFetchInterval(UIApplication.backgroundFetchIntervalMinimum)
```

This line will ask iOS to wake the app up at a minimum interval. This means that the app is woken up as often as iOS will allow it to be woken up. It's impossible to predict how often this will be since iOS throttles this interval as it sees fit. If your app often has new data available or is used many times throughout the day, the odds are that your app will be woken up more often than it would be if you rarely have new data available or if the user opens your app once every couple of days.

Alternatively, you could set the fetch interval to `UIApplication.backgroundFetchIntervalNever` if you want to prevent your app from being woken up at all, or you can use a custom value. If you provide a custom value, iOS will attempt to honor your custom interval, but again, there are no guarantees with background fetch because the system remains in charge concerning the intervals that are used.

The only thing iOS does guarantee is that your app is not woken up more often than the value you specified. So, let's say you define a custom interval for your app to be woken up once every three hours, iOS won't wake your app more often than that, but it's possible that your app is woken up only once every eight hours. This depends on several factors that are not transparent to you as a developer.

Your app is now ready to fetch data in the background. Let's go ahead and implement the fetching of data.

Updating movies in the background

The final step in enabling background fetch for **MustC** is to add the `application(_:performFetchWithCompletionHandler:)` method to `AppDelegate`. As explained before, this method is called by iOS whenever the app is woken up in the background, and it allows you to perform an arbitrary amount of work. Once the app is done performing its task, it must call the completion-handler that was passed to this method.

Upon calling the completion-handler, you should inform iOS about the results of the operations you performed. It's important to correctly report this status because background fetch is intended to improve the user experience. If you falsely report to iOS that you have new data all the time, so your app is woken up more often, you're degrading the user experience. You should trust the system to judge when your app is woken up. It's in the best interest of your users, their battery life, and ultimately your app to not abuse background fetch.

To efficiently implement background fetch, you will work through the following steps:

1. Updating the data model so you can query the movie database more efficiently.
2. Refactoring the existing code to use the improved data model.
3. Implementing background fetch with the existing helper struct.

The first two steps are not directly tied to implementing background fetch, but they do illustrate that an efficient background fetch strategy may involve refactoring some of your app's existing code. Remember, there is nothing wrong with refactoring old code to implement a new feature. Both the new feature and the old code will usually benefit from refactoring.

Updating the data model

The data model you currently have associates movies with a single family member. This means that the app potentially stores the same movie over and over again. When we were only storing data, this wasn't that big of a deal. This wasn't a big deal when you first implemented your database. However, now that the app will query the movie database in a background fetch task, it would be a waste of resources to ask the movie database for the same ratings multiple times. Also, you most certainly don't want to use the movie database search API in the same way you did before; you should refer to the movie you want as precisely as possible.

To facilitate this, the relationship between movies and family members must be changed to a many-to-many relationship. You'll also add a new field to the movie entity: `remoteId`. `remoteId` will hold the identifier the movie database uses for a particular movie so it can be used directly in later API calls.

Open the model editor in Xcode and add the new `remoteId` property to **Movie**. Make sure that it's a 64-bit integer and that it's optional. Also, select the **familyMember** relationship and change it to a **To Many** relationship in the sidebar. It's also a good idea to rename the relationship to **familyMembers**, since it's now related to more than one family member. Your model should resemble the model in the following screenshot:

The screenshot shows the Xcode Model Editor with the 'Movie' entity selected in the left sidebar under 'ENTITIES'. The main area displays three sections: 'Attributes', 'Relationships', and 'Fetched Properties'. In the 'Attributes' section, there are three attributes: 'popularity' (Double), 'remotelyId' (Integer 64), and 'title' (String). In the 'Relationships' section, there is one relationship named 'familyMembers' of type 'FamilyMember', which is inverse to the 'movies' relationship. The 'Fetched Properties' section is currently empty. To the right of the entity configuration, there are several tabs: 'Entity', 'Class', 'Constraints', 'Spotlight', 'User Info', and 'Versioning'. The 'Entity' tab is active, showing details like 'Name: Movie', 'Module: Global namespace', and 'Codegen: Class Definition'.

Great, the model has been updated. There's still a lot of work to be done though. Because the name and nature of the family member relationship were changed, the project won't compile. Make the following modifications to the `managedObjectContextDidChange(_:)` method in `MoviesViewController.swift`:

```
if let updatedObjects = userInfo[NSUpdatedObjectsKey] as? Set<Movie> {
    for object in updatedObjects {
        if let familyMember = self.familyMember,
           let familyMembers = object.familyMembers,
           familyMembers.contains(familyMember) {

            tableView.reloadData()
            break
        }
    }
}
```

There is just one more model-related change that you should incorporate. To efficiently search for an existing movie or create a new one, an extension to the `Movie` model should be created. Create a new group called **Models** in your project and add a new Swift file named `Movie.swift` to it. Finally, add the following implementation to the file:

```
import CoreData

extension Movie {
    static func find(byName name: String, orCreateIn moc: NSManagedObjectContext) -> Movie {
        let predicate = NSPredicate(format: "title ==[dc] %@", name)
        let request: NSFetchedResultsController<Movie> = Movie.fetchRequest()
        request.predicate = predicate

        guard let result = try? moc.fetch(request)
            else { return Movie(context: moc) }

        return result.first ?? Movie(context: moc)
    }
}
```

The preceding code queries Core Data for an existing movie with the provided name. The movies are matched case-insensitive by passing `[dc]` to the `==` operator in `NSPredicate`. It's important that this lookup is case-insensitive because people might write the same movie name with different capitalizations. If no movies could be found, or if the results come back empty, a new movie is created. Otherwise, the first, and presumably the only, result that Core Data has is returned. This wraps up the changes that need to be made to the app's data layer.

Refactoring the existing code

The existing code compiles, but it's not optimal yet. `MovieDBHelper` doesn't pass the movie's remote ID to its callback, and the movie-insertion code doesn't use this remote ID yet. When the user wants to save a new movie, the app still defaults to creating a new movie instead of using the handy new helper you just wrote to avoid data duplication. You should update the code, so the callback is called with the fetched remote ID.

Let's update MovieDBHelper first. Replace the following lines in the `fetchRating(forMovie:callback:)` method:

```
typealias MovieDBCallback = (Int?, Double?) -> Void
let apiKey = "d9103bb7a17c9edde4471a317d298d7e"

func fetchRating(forMovie movie: String, callback: @escaping
MovieDBCallback) {
    guard let searchUrl = url(forMovie: movie) else {
        callback(nil, nil)
        return
    }

    let task = URLSession.shared.dataTask(with: searchUrl) { data,
response, error in
        var rating: Double? = nil
        var remoteId: Int? = nil

        defer {
            callback(remoteId, rating)
        }

        let decoder = JSONDecoder()

        guard error == nil, let data = data,
              let lookupResponse = try?
decoder.decode(MovieDBLookupResponse.self, from: data),
              let movie = lookupResponse.results.first
        else { return }

        rating = movie.popularity
        remoteId = movie.id
    }

    task.resume()
}
```

These updates change the callback-handler, so it takes both the remote ID and the rating as parameters. A variable to hold the remote ID is added and incorporated into the callback. With this code, `MovieDBHelper` is entirely up to date.

You should also update the response struct, so the `MovieDBMovie` struct includes the ID from the API response:

```
struct MovieDBMovie: Codable {
    let popularity: Double?
    let id: Int?
}

moc.persist {
    // 1
    let movie = Movie.find(byName: name, orCreateIn: moc)

    // 2
    if movie.title == nil || movie.title?.isEmpty == true {
        movie.title = name
    }

    let newFavorites: Set <AnyHashable> =
familyMember.movies?.adding(movie) ?? [movie]
    familyMember.movies = NSSet(set: newFavorites)

    let helper = MovieDBHelper()
    helper.fetchRating(forMovie: name) { remoteId, rating in
        guard let rating = rating,
              let remoteId = remoteId
        else { return }

        moc.persist {
            movie.popularity = rating
            movie.remoteId = Int64(remoteId)
        }
    }
}
```

First, the preceding code either fetches an existing movie or creates a new one with the `find(byName:orCreateIn:)` method you just created. Next, it checks whether the returned movie already has a name. If it doesn't have a name yet, the name is set. Also, if it does have a name, you can safely assume you were handed an existing movie object, so there is no need to set the name. Next, the rating and ID are fetched, and the corresponding properties on the movie object are set to their correct values in the callback.

This is all the code needed to prepare **MustC** for background fetch. Let's implement this feature now.

Updating movies in the background

Almost all of the building blocks involved with implementing background fetch are in place. All you need now is a way to fetch movies from the movie database using their remote ID instead of using the movie database search API.

To enable this way of querying movies, another fetch method is required on `MovieDBHelper`. The simplest way to do this would be to copy and paste both the fetch and URL-building methods and adjust them to enable fetching movies by ID. While it might be the easiest way to go about this, it isn't the best idea. If you add another fetch method or require more flexibility later, you will be in trouble. It's much better to refactor the movie db helper so it's better suited to multiple types of requests.

Preparing the helper struct

To maintain a clear overview of the available API endpoints, you can add a nested enum to `MovieDBHelper`. Doing this will make other parts of the code more readable, help to avoid errors, and abstract away code duplication. An associated value will be used on the enum to hold onto the ID of a movie. This is quite convenient because the movie ID is part of the API endpoint.

Add the following code inside of the `MovieDBHelper` struct:

```
static let apiKey = "d9103bb7a17c9edde4471a317d298d7e"

enum Endpoint {
    case search
    case movieById(Int64)

    var urlString: String {
        let baseUrl = "https://api.themoviedb.org/3/"

        switch self {
        case .search:
            var urlString = "\u{2028}(baseUrl)search/movie/"
            urlString =
                urlString.appending("?api_key=\u{2028}(MovieDBHelper.apiKey)")
            return urlString
        case let .movieById(movieId):
            var urlString = "\u{2028}(baseUrl)movie/\u{2028}(movieId)"
            urlString =
                urlString.appending("?api_key=\u{2028}(MovieDBHelper.apiKey)")
            return urlString
        }
    }
}
```

```
        }
    }
}
```

Note that the `apiKey` constant has been changed from an instance property to a static property. Making it a static property makes it available to be used inside of the nested `Endpoint` enum. Note that the value associated with the `movieById` case is `Int64` instead of `Int`. This is required because the movie ID is a 64-bit integer type in Core Data.

With this new `Endpoint` enum in place, you can refactor the way you build the URLs as follows:

```
private func url(forMovie movie: String) -> URL? {
    guard let query = movie.addingPercentEncoding(withAllowedCharacters:
        .urlHostAllowed)
    else { return nil }

    var urlString = Endpoint.search.urlString
    urlString = urlString.appending("&query=\\"(query)"")
    return URL(string: urlString)
}

private func url(forMovieId id: Int64) -> URL? {
    let urlString = Endpoint.movieById(id).urlString
    return URL(string: urlString)
}
```

The `url(forMovie:)` method was updated to make use of the `Endpoint` enum. The `url(forMovieId:)` method is new and uses the `Endpoint` enum to obtain a movie-specific URL easily.

When you fetch data, there are some pieces of code that you will have to write regardless of the URL that will ultimately be used to fetch data. When you look at the `fetchRating(forMovie:)` method, there are a couple of things that apply to both endpoints you will eventually use to retrieve movie information. The following list is an overview of these things:

- Checking whether we're working with a valid URL
- Creating the data task
- Extracting the JSON
- Calling the callback

If you think about it, the only real difference is the API response object that is used. When you examine the JSON responses returned by the search API and fetching a movie by ID, the difference is that the search API returns an array of movies where you're interested in the first result. The movie-by-id API returns the correct movie as part of the root object.

With this in mind, the refactored code should be able to retrieve the desired data using just a URL, a data-extraction strategy, and a callback. Based on this, you can write the following code:

```
// 1
typealias IdAndRating = (id: Int?, rating: Double?)
typealias DataExtractionStrategy = (Data) -> IdAndRating

// 2
private func fetchRating(fromUrl url: URL?, withExtractionStrategy
extractionStrategy: @escaping DataExtractionStrategy, callback:
@escaping MovieDBCallback) {
    guard let url = url else {
        callback(nil, nil)
        return
    }

    let task = URLSession.shared.dataTask(with: url) { data, response,
error in
        var rating: Double? = nil
        var remoteId: Int? = nil

        defer {
            callback(remoteId, rating)
        }

        guard error == nil
        else { return }

        guard let data = data
        else { return }

        // 3
        let extractedData = extractionStrategy(data)
        rating = extractedData.rating
        remoteId = extractedData.id
    }

    task.resume()
}
```

There is quite a lot going on in the preceding snippet. Most of the code will look familiar, but some of the details might be new. Let's go over the comments in this code one by one:

1. This part defines a couple of type aliases that will make the code a little bit more readable. The first alias is for a named tuple that contains the movie's ID and rating. The second alias defines the signature for the data extraction closure.
2. The second highlight is for `fetchRating(fromUrl:withExtractionStrategy:callback:)`. This method will be used to obtain movie data. It's marked private because it's not very useful to call this method directly. Instead, it will be called by two methods that will be implemented soon.
3. The data for the movie is extracted from the raw data by passing the raw data to the data extraction closure.

Let's use this method to implement both the old way of fetching a movie through the search API and the new way that uses the movie ID to request the resource directly, as follows:

```
func fetchRating(forMovie movie: String, callback: @escaping MovieDBCallback) {
    let searchUrl = url(forMovie: movie)
    let extractData: DataExtractionStrategy = { data in
        let decoder = JSONDecoder()

        guard let response = try?
            decoder.decode(MovieDBLookupResponse.self, from: data),
              let movie = response.results.first
        else { return (nil, nil) }

        return (movie.id, movie.popularity)
    }

    fetchRating(fromUrl: searchUrl, withExtractionStrategy: extractData,
               callback: callback)
}

func fetchRating(forMovieId id: Int64, callback: @escaping MovieDBCallback) {
    let movieUrl = url(forMovieId: id)
    let extractData: DataExtractionStrategy = { data in
        let decoder = JSONDecoder()

        guard let movie = try?
```

```
decoder.decode(MovieDBLookupResponse.MovieDBMovie.self, from: data)
    else { return (nil, nil) }

    return (movie.id, movie.popularity)
}

fetchRating(fromUrl: movieUrl, withExtractionStrategy: extractData,
callback: callback)
}
```

The code duplication is minimal in these methods, which means that this attempt at refactoring the code was a huge success. If you add new ways to fetch movies, all you will need to do is obtain a URL, define how to retrieve the data you're looking for from the `data` object, and finally, kick off the fetching.

You're now finally able to fetch movies using their ID without duplicating a lot of code. The final step in implementing the background update feature is to implement the code that updates movies. Let's go!

Updating the movies

The process of updating movies is a strange one. As you saw earlier, network requests are performed asynchronously, which means that you can't rely on the network request being finished by the time a function is finished executing. Because of this, a callback is used. The callback is then called when the network request is done.

But what happens if you need to wait for multiple requests? How do you know that all requests to update movies have been completed? Since the movie database doesn't allow developers to fetch multiple movies at once, a bunch of requests must be made. When all of these requests are complete, the background fetch `completionHandler` should be called with the result of the operation.

To achieve this, you can make use of the grand central dispatch. More specifically, you can use a dispatch group. A dispatch group keeps track of an arbitrary number of tasks, and it won't consider itself as completed until all of the tasks that are added to the group have finished executing.

This behavior is precisely what's needed to wait for all movies to be updated. Whenever we fetch a movie from the network, you can add a new task to the dispatch group that will be marked as completed once the underlying movie is updated. Finally, when all of the movies are updated, `completionHandler` can be called to inform iOS about the result of the background fetch. Let's take a step-by-step look at how to achieve this behavior. Start by adding the following code to `AppDelegate`:

```
func application(_ application: UIApplication,  
performFetchWithCompletionHandler completionHandler: @escaping  
(UIBackgroundFetchResult) -> Void) {  
  
    let fetchRequest: NSFetchedResultsController<Movie> = Movie.fetchRequest()  
    let managedObjectContext = persistentContainer.viewContext  
    guard let allMovies = try? managedObjectContext.fetch(fetchRequest)  
    else {  
        completionHandler(.failed)  
        return  
    }  
}
```

This first part of the background fetch is relatively straightforward. All it does is retrieve all movie objects from the database. If this fails, `completionHandler` is called with a `.failed` status.

All the following code snippets should be added to the `application(_:performFetchWithCompletionHandler:)` method inside of `AppDelegate` in the same order as they are presented. A full overview of the implementation will be provided at the end:

```
let queue = DispatchQueue(label: "movieDBQueue")  
let group = DispatchGroup()  
let helper = MovieDBHelper()  
var dataChanged = false
```

These lines create a dispatch queue and a dispatch group. The dispatch queue represents the background thread on which the fetch operations will be executed. Next, add the following snippet:

```
for movie in allMovies {  
    queue.async(group: group) {  
        group.enter()  
        helper.fetchRating(forMovieId: movie.remoteId) { id, popularity in  
            guard let popularity = popularity,  
                  popularity != movie.popularity else {  
                group.leave()  
                return  
            }  
            movie.popularity = popularity  
            dataChanged = true  
        }  
    }  
}  
queue.notify(queue: group)  
group.notify(queue: queue) {  
    completionHandler(.success)  
}
```

```
        }

        dataChanged = true

        managedObjectContext.persist {
            movie.popularity = popularity
            group.leave()
        }
    }
}
```

This part of the implementation loops through the fetched movies. For every movie, a task is added to the dispatch queue and `group.enter()` is called to tell the dispatch group that a new task has just been added. The next step is to fetch the rating. If this fails, `group.leave()` is called to tell the dispatch group that this task has finished. If the data was retrieved successfully, the movie is updated with the fetched rating. Once the managed object has persisted the changes, `group.leave()` is called, and the operation has finished.

The next and final snippet that must be added executes when all the tasks in the queue are performed; at this point, the code should check whether new data has been fetched by reading the `dataChanged` property, and based on this property, call `callbackHandler`:

```
group.notify(queue: DispatchQueue.main) {
    if dataChanged {
        completionHandler(.newData)
    } else {
        completionHandler(.noData)
    }
}
```

The `group.notify` method takes a queue and a block of code that it should execute. The queue is set to the main queue, which means that the code inside of the block is performed on the main queue. Then, the `dataChanged` variable is read, and `completionHandler` is called accordingly.

As promised, the full implementation for `application(_:performFetchWithCompletionHandler:)` is as follows:

```
func application(_ application: UIApplication,
performFetchWithCompletionHandler completionHandler: @escaping
(UIBackgroundFetchResult) -> Void) {

    let fetchRequest: NSFetchedResultsController<Movie> = Movie.fetchRequest()
```

```
let managedObjectContext = persistentContainer.viewContext
guard let allMovies = try? managedObjectContext.fetch(fetchRequest)
else {
    completionHandler(.failed)
    return
}

let queue = DispatchQueue(label: "movieDBQueue")
let group = DispatchGroup()
let helper = MovieDBHelper()
var dataChanged = false

for movie in allMovies {
    queue.async(group: group) {
        group.enter()
        helper.fetchRating(forMovieId: movie.remoteId) { id, popularity
in
            guard let popularity = popularity,
                  popularity != movie.popularity else {
                group.leave()
                return
            }

            dataChanged = true

            managedObjectContext.persist {
                movie.popularity = popularity
                group.leave()
            }
        }
    }
}

group.notify(queue: DispatchQueue.main) {
    if dataChanged {
        completionHandler(.newData)
    } else {
        completionHandler(.noData)
    }
}
```

To test whether background fetching is working as expected, you can build and run your app. Then, add a new movie, so you have a movie for which the ID is stored. Finally, you can use the debug menu item in Xcode that's at the top of your screen to simulate a background refresh. This will trigger a background refresh.

Summary

This chapter opened with an overview of background fetch, how it works, and how it benefits your users. You also learned about the prerequisites and best practices for this feature. After establishing this basic understanding, you learned how background fetch works in concert with your application. Then, we continued to implement the required permissions and asked iOS to wake up the **MustC** application periodically so we could update the movies' ratings.

Once you did this, you had to refactor a good portion of the application to accommodate the new feature. It's important to be able to recognize scenarios where refactoring is a good idea, especially if it enables the smooth implementation of a feature later on. It also demonstrates that you don't have to think about every possible scenario for your code every time you implement a feature. After refactoring the application, you were finally able to implement background-fetching behavior. To do this, you quickly glanced over dispatch groups and how they allow you to group an arbitrary amount of asynchronous work together and be notified when all of the tasks in a group are completed.

Now that you know how to fetch data from the network, store it locally, and update it in the background, wouldn't it be great if you could sync data across devices and automatically update it in the background as soon as new data is available? That's precisely what the next chapter is all about!

Test your knowledge

1. For how long can you perform work when your app is woken up for background refresh?
 - a) About two minutes.
 - b) There is no way to be sure.
 - c) You have no time limit.
2. How do you enable background fetch for your app?
 - a) By requesting it in `AppDelegate`.
 - b) By enabling the Background Refresh capability.
 - c) Both.

3. Which statement about the minimum refresh interval is true?
 - a) Your app will be woken up every time the interval passes.
 - b) iOS will determine when it's the best time to wake up your app. It will be no more often than you specify.
 - c) iOS will ask your app whether it has any work to do when the interval passes.
4. What did you use a dispatch group for in this chapter?
 - a) To take work off the main thread.
 - b) To be notified when all network calls were done.
 - c) To make sure all network calls are fired at the same time.
5. When should you refactor your code?
 - a) When you get frustrated with your code.
 - b) When you find strange bugs.
 - c) Any time you think you can improve something.
6. What do you call the ID that is passed to the movieById case on the MovieDBHelper.Endpoint enum?
 - a) Associated value.
 - b) Attached value.
 - c) Enum value.

Further reading

- Chapter 28, *Offloading Tasks with Operations and GCD*
- Updating your App with Background App Refresh: https://developer.apple.com/documentation/uikit/core_app/managing_your_apps_life_cycle/preparing_your_app_to_run_in_the_background/updating_your_app_with_background_app_refresh

15

Syncing Data with CloudKit

Storing data locally in Core Data is pretty sweet, but storing data locally on a device has its limitations. When a user gets a new device or uninstalls your app, their database is gone. This could be very inconvenient for the user because they might want to hold on to their data somehow.

Another inconvenience of a local database is that the database isn't shared across a user's devices. To work around this problem, you could implement your own backend server and store your user's data there. However, this comes with great responsibility because you would have to ensure that your user's data is protected and stored securely, not to mention the amount of effort and costs that will go into building and maintaining your own backend.

If you're looking for an excellent way to synchronize data across a user's device without having to build your own backend or invest vast amounts of time into storing data securely, **CloudKit** is precisely what you need. In this chapter, you will learn how you can set up a simple data store on Apple's CloudKit service to allow data-sharing between a user's devices and even how you can share data across multiple users!

In this chapter you will learn about the following topics:

- Getting familiar with CloudKit
- Storing and retrieving data with CloudKit
- Combining CloudKit and Core Data

By the end of this chapter, you will know everything you need to know about implementing CloudKit in an application and keeping a local Core Data database in sync with the remote data.

Getting familiar with CloudKit

CloudKit is a service that Apple created as a simple tool for application developers to store data on a remote server. The beauty of this service is that any user with an iCloud account can be identified automatically by CloudKit, meaning you can begin connecting data to your user immediately without having them first sign up for your service.

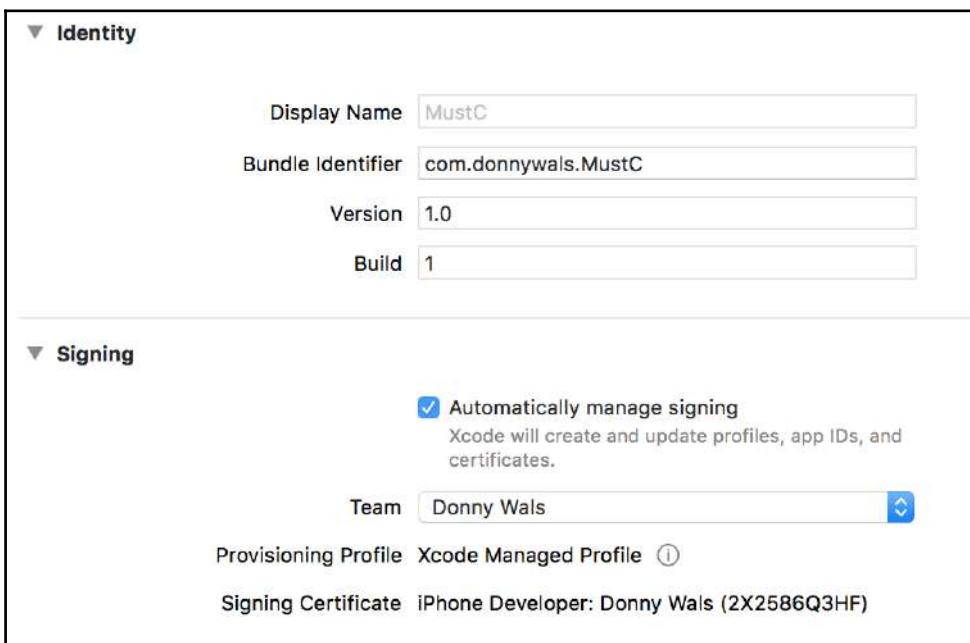
Signup screens are often considered to be dealbreakers for users because they don't want to share their email address with certain apps, or just because they think signing up for a particular app isn't worth the effort. With CloudKit, you can remove all this friction because users are discretely logged into your app using the iCloud account that they already use on their device.

When you add CloudKit to your app, you will find that it offers a lot of great functionalities to help you keep the number of web requests as low as possible, and you can even have CloudKit notify your app when something changes on the remote server even when your app isn't running. Apple uses features such as these extensively in their own applications to try to make sure that their apps provide a seamless experience between devices.

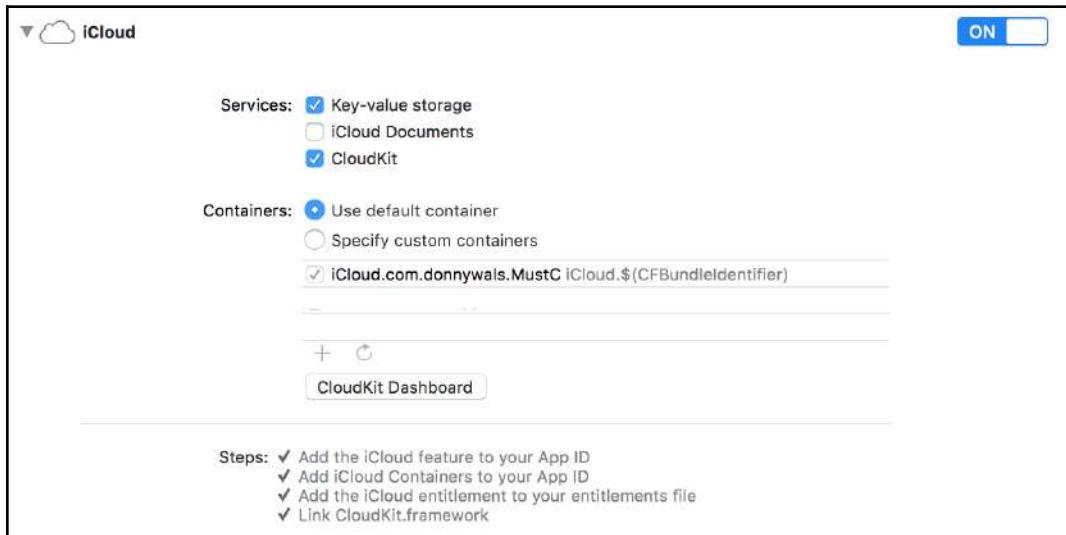
To add CloudKit to the **MustC** app, you don't need to do a lot of work. All you need to do is make sure your project is set up correctly and enable the required capabilities for your app.

Adding CloudKit to your project

First, make sure the Xcode project is set up correctly. Note that the project files in this book's code bundle have been modified in preparation of this chapter. If you want to follow along with the steps in this chapter, make sure to grab the **MustC** starter project for this chapter from the code bundle. You need to have a unique bundle identifier for the **MustC** app. The code bundle that goes with this book uses a bundle identifier of `com.donnywals.MustC`. Make sure that you change this to something unique. Also, make sure that you have selected a valid developer team under Xcode's signing options. Refer to the following screenshot to see an example of what your **General** settings should look like:



Next, open the **Capabilities** tab in the project settings. Scroll down until you see the **iCloud** capability. Make sure to toggle it so it's turned on and expand the iCloud section. Check the **CloudKit** checkbox to enable CloudKit for your application, as shown in the following screenshot:



Toggling the iCloud capability has added an entitlements file to your project. This file is coupled to your Xcode signing identity and it makes sure that your app is allowed to use certain capabilities. You should not make manual changes to this file because Xcode will automatically ensure that it matches the **Capabilities** tab in your project settings.

Note that enabling iCloud and CloudKit also turns on the Push Notifications entitlement. This entitlement was added to your application so it can receive notifications about changes in your CloudKit database in the background. You'll learn more about this later in the chapter.

Technically this is all you need to do to add CloudKit to your app. Before you learn how to perform read and write operations on your CloudKit database, you should familiarize yourself with the heart of your CloudKit backend, the dashboard.

Exploring the CloudKit dashboard

If you look at the **Capabilities** tab in your project settings, there is a **CloudKit Dashboard** button underneath your iCloud capability configuration. Click this button or navigate to <https://icloud.developer.apple.com/dashboard/> in your web browser to go to the CloudKit dashboard. After logging into the dashboard, you'll see your developer teams and the containers that you have set up for each team.



Sometimes it takes a little while for a new container to show up in the CloudKit dashboard. If your project doesn't show up right away, don't worry. Just take a five-minute break and then check whether your container shows up.

A CloudKit container uses a unique identifier that is based on your app's bundle identifier. To explore your CloudKit container, click it. You should see a screen that resembles the following screenshot:

The screenshot shows the CloudKit Dashboard interface. At the top, the URL is 'iCloud.com.donnywals.MustC'. On the right, there is a dropdown menu for 'DONNY WALS'. Below the URL, a message says 'Choose an area in an environment to view:'. The main area is divided into two sections: 'Development' (blue header) and 'Production' (green header). Both sections contain five items: 'Data', 'Logs', 'Telemetry', 'Public Database Usage', and 'API Access'. Each item has a small icon and a brief description. At the bottom of each section are three buttons: 'Reset...', 'Environment Settings...', and 'Deploy to Production...'. In the 'Production' section, the 'Environment Settings...' button is highlighted with a green border.

This screen is the starting point for your container. Every container has a development environment and a production environment. This is convenient because it allows you to experiment as much as you'd like in your development database without disrupting the production database where your user's data will live.

From this screen, you can monitor what your container is doing, how much it's used, and you can even access logs that help you with debugging issues if you encounter them. For now, go to the **Data** section so you can explore your CloudKit container's data store.

Understanding your CloudKit database

In order to efficiently use CloudKit to store data, you must understand how it stores data and what terminology is used for the different components that are used throughout your database. Every CloudKit container comes with a couple of different databases:

- A private database for each user
- A shared database for each user
- A public database that's shared for all users

It's important that you use the correct database whenever you write data to CloudKit, you wouldn't want to accidentally store some sensitive user information in the public database because that could potentially expose the information to all users of your app. You should only use the public database for data that is suited for *all* users of an application.

For instance, if you're building a news app, it would make sense to store all the news and possibly comments on news articles in the public database for your container. However, saving a user's to-do list should be stored in the private database because no other users have any business with another user's to-do list.

One exception to this is when a user shares their list or individual items with another user. This shared data can be stored in the shared database where only users that have explicitly gained access to the records are allowed to access them.

Now that you know about the different types of databases, let's have a look at the data page itself to see what kind of options you have to manage data there.

The leftmost tab in the data manager is the **Zones** tab. You can use zones to logically group certain parts of your database together. For instance, you could create zones for data that originated from different origins. The private and public databases come with a default zone. You cannot create custom zones for the public database so all public data is automatically stored in the default zone. For the private and shared databases, you can create custom zones.

Note that the zones page has the option to **Fetch zone changes since...** This is a performance optimization that makes sure apps can load a minimal amount of data whenever they want to synchronize with the data from CloudKit. If you want to add a custom zone, you can do so by clicking on the **Create new Zone...** button. While you're there, create a new zone for the private database and name it `moviesZone`. This is where you'll store the data from the **MustC** app later.

The second tab, **Records**, is more interesting. On this tab, you can perform queries on the data that is stored in your databases and zones. This allows you to explore your data and make sure that everything is saved as you would expect. For privacy reasons, you can only see the data for your own private and shared database and the public database. You can never access a user's private database.

The records tab provides a window into your CloudKit database. Note that the records page also has a **Changes** tab. You can use this tab to retrieve changes to all records in the selected zone. Note that you can't do this for the default zone, so if you want to use this functionality in your app, you will need to create a custom zone on the **Zones** tab.

The third tab is the **Record Types** tab. This is where you define the objects you want to store in your CloudKit database and what properties they have. Note that record types are not connected to a zone. This means that you can write a certain record to any zone you'd like. Keep in mind though that records from different zones cannot reference each other. So if you consider the **MustC** app, a family member and their favorite movies would have to exist in the same zone. If this isn't the case, the objects are not allowed to reference each other in CloudKit.

To prepare for storing data later, you should create your record types now. Click the **Create New Type** button and name it `FamilyMember`. Add a field to it by clicking the **Add field** button and call the field **name**. The type should be **String**. Add another field and call it **movies**, give it a type of **Reference (list)**. The following screenshot shows what your fields should look like:



Click the **Create Record Type** button to store your record type and create a second record type for the movie records. Use **Movie** as the name for your record type and give it three properties, named **title**, which should be a String, **rating** (Double), and a **remoteId** (Int64). Make sure to save this record type as well.

After you save a record type, CloudKit automatically adds some extra metadata properties to it. These properties are used internally by CloudKit to figure out what objects changed and when. You can't remove or modify these properties because CloudKit needs them.

The fourth tab in the dashboard is the **Indexes** tab. This tab is used to specify which properties of a record will be optimized for querying. If you don't specify any indexes, you won't be able to easily fetch all records from your CloudKit database. Typically you will want to at least add an index for the **recordName** property. To do this, select your record type and click the **Add Index** button. Select the **recordName** property in the first dropdown and **QUERYABLE** from the second. Do this for both record types you created earlier.

The fifth tab is **Subscriptions**. This is where you can see the subscriptions that exist for a particular database. You can only look at the subscriptions for your own private and shared database, and the public database. You can use this tab to check whether you are correctly subscribing apps to database updates.

The next tab is **Subscription types**. There isn't much to do here other than explore the automatically-generated subscription types and check whether they match what you expect to find there.

The last tab is the **Security Roles** tab. This tab allows you to manage what permissions users have for record types in the public database. This allows you to limit the actions that a certain user can perform. For instance, you might want to allow all users to read news articles from the public database but restrict writing new articles to the database to your administrative users. You can manage all this in the security roles tab. The default values will suffice for the **MustC** app.

Let's go back to your container overview page to discuss the other available sections briefly.

Exploring the rest of your container

From your container overview page, you can gain insights into several interesting and helpful metrics. You can use the **Logs** page to see real-time logs about what's going in your application. You can use these logs to make sure specific data is sent and received as you would expect, or to troubleshoot problems your users are having. Note that you can't see the private data that your users are storing, but you can look at the operations they are trying to perform on the CloudKit database.

The **Telemetry** page gives you insight into what's going on inside of your application. You can see performance-related statistics and learn how your CloudKit database is used. You can use this page together with **Public Database Usage** to understand where your implementation could be improved and how users use your app.

The last section is **API Access**. This section is relevant when you want to use the CloudKit store outside of iOS, for instance from a web page or web server.

This information should cover the most interesting bits of the CloudKit dashboard. Let's see how you can make your app communicate with CloudKit.

Storing and retrieving data with CloudKit

CloudKit is meant to be used as a backend for your app that syncs across devices and provides a seamless experience for users. This section will go over the principles that you should keep in mind when implementing CloudKit in your app. By the end of this section, you will know precisely how storing, retrieving, and updating data in CloudKit works. You will then go on to implement CloudKit in the **MustC** app by combining it with the existing Core Data database.

Communicating with CloudKit for the first time

When your app launches for the very first time, it should immediately make itself known to CloudKit so it can subscribe to changes. Technically you don't have to do this because you can implement your own logic to retrieve data from CloudKit as you need it. However, an efficient CloudKit implementation will work flawlessly when there is no internet connection, which means that it's probably a good idea to store all data locally by writing it to a database such as Core Data.

The recommended way to subscribe to database changes is to keep track of whether your app is already subscribed locally. You can keep track of this by storing a variable in the local `UserDefault`s store. `UserDefault`s is perfect for storing variables such as this. Open `CloudStore.swift` in the **MustC** project and add the following computed property to it:

```
var isSubscribedToDatabase: Bool {
    get { return UserDefaults.standard.bool(forKey:
        "CloudStore.isSubscribedToDatabase") }
    set { UserDefaults.standard.set(newValue, forKey:
        "CloudStore.isSubscribedToDatabase") }
}
```

This property proxies the standard `UserDefault`s through its custom getters and setters. A construction such as this is very convenient since it makes using this property simple and straightforward even though there's a bit more logic going on behind the scenes.

With the `isSubscribedToDatabase` property in place, you should also implement a method that's responsible for subscribing the app to changes in the CloudKit database. Add the following implementation to `CloudStore.swift`:

```
extension CloudStore {
    func subscribeToChangesIfNeeded(_ completion: @escaping (Error?) ->
    Void) {
        // 1
        guard isSubscribedToDatabase == false else {
            completion(nil)
            return
        }

        // 2
        let notificationInfo = CKSubscription.NotificationInfo()
        notificationInfo.shouldSendContentAvailable = true

        // 3
        let subscription = CKDatabaseSubscription(subscriptionID:
            "private-changes")
        subscription.notificationInfo = notificationInfo

        // 4
        let operation =
            CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
            subscriptionIDsToDelete: [])

        // 5
    }
}
```

```
        operation.modifySubscriptionsCompletionBlock = { [unowned self]
    subscriptions, subscriptionIDs, error in
        if error == nil {
            self.isSubscribedToDatabase = true
        }

        completion(error)
    }

    // 6
    privateDatabase.add(operation)
}
}
```

Let's go over the preceding code step by step, so you understand what happens in this snippet:

1. If the application has already subscribed to changes, there is no need to subscribe again. This means that you can call the completion closure without any errors.
2. When there are changes in the database, CloudKit will send a silent push notification to the app. Silent push notifications can be used to let the app know that new data is available. This mechanism is very similar to background fetch, which is covered in [Chapter 14, *Being Proactive with Background Fetch*](#).
3. Every subscription in CloudKit is created as an instance of `CKDatabaseSubscription`. The notification settings that were configured in step two are attached to the database subscription.
4. This step creates an instance of `CKModifySubscriptionsOperation`. CloudKit uses a concept called `Operations` to manage the operations that you want the database to execute. Subscribing to changes is an example of such an operation, but fetching changes from the database or sending data to the database are also examples of operations. You can read more about `Operations` in [Chapter 28, *Offloading Tasks with Operations and GCD*](#).
5. This step defines a closure that is called when the operation to subscribe to changes has finished. If this operation completes without error, the app is subscribed to changes and the status is persisted to `UserDefault`s. Every CloudKit operation has one or more of these closure callbacks that you can implement to respond to certain events that can occur during an operation.

6. The last step is to add the operation to the private database. Since all data will be stored in the private moviesZone that you created earlier, the appropriate database to use for the subscription is the private database.

Your app is now subscribed to changes from the CloudKit database. The next step is to make sure your app receives and handles updates from the CloudKit server.

Listening for changes in the database

When new data is available on the CloudKit server, it sends a notification to all subscribed apps. A great way to handle these notifications is to add a method to `CloudStore` that should be called when there are new changes in the CloudKit database. Add the following extension to `CloudStore.swift`:

```
extension CloudStore {  
    func handleNotification(_ dict: [String: AnyObject],  
                           completionHandler: @escaping (UIBackgroundFetchResult) -> Void) {  
        let notification =  
            CKNotification(fromRemoteNotificationDictionary: dict)  
  
        if notification.subscriptionID == "private-changes" {  
            fetchDatabaseChanges { error in  
                if error == nil {  
                    completionHandler(.newData)  
                } else {  
                    completionHandler(.failed)  
                }  
            }  
        }  
    }  
}
```

This method takes a dictionary and a completion-handler as its arguments. Note that the completion-handler has one argument, which is `UIBackgroundFetchResult`. As mentioned before, CloudKit sends a silent push notification when data has changed to trigger a process that's very similar to background fetch.

The previous method uses the notification dictionary to create a `CKNotification` instance and then checks whether the notification originated through the subscription for the private database. If this is the case, the changes should be fetched and ultimately get persisted to the local database.

When you subscribe to database changes, CloudKit will notify your app when any data in the database changes. When you fetch database changes, CloudKit will not tell you what records have changed straight away. Instead, CloudKit will send you the database zones that have changed. A flexible CloudKit setup should be able to handle incoming data from different zones, which means that you might add a new zone to your database and the app should be able to handle this.

If your app receives a notification to inform it of new changes, CloudKit won't provide any additional information. Instead, it is your job to ask CloudKit for the relevant changes. Remember that you were able to request only data that has changed since a specific moment in the CloudKit dashboard? The same mechanism is used when you fetch changes for your app.

Retrieving changes from CloudKit

To fetch changes that occurred after a certain point in time, you need to a change token along with your fetch operations. Add the following computed property to the `CloudStore` class so you can keep track of the change token that represents the last time the app fetched changes from CloudKit:

```
var privateDatabaseChangeToken: CKServerChangeToken? {
    get { return UserDefaults.standard.serverChangeToken(forKey:
        "CloudStore.privateDatabaseChangeToken") }
    set { UserDefaults.standard.set(newValue, forKey:
        "CloudStore.privateDatabaseChangeToken") }
}
```

This computed property uses a helper method on `UserDefault`s to store `CKServerChangeToken`.



If you're curious to see how the computed property is implemented, make sure to look at the `UserDefault.swift` file in the project from the code bundle.

Add the following code to `CloudStore.swift` to implement the logic to fetch database changes:

```
extension CloudStore {
    func fetchDatabaseChanges(_ completionHandler: @escaping (Error?) ->
        Void) {
        // 1
        let operation =
```

```
CKFetchDatabaseChangesOperation(previousServerChangeToken:  
privateDatabaseChangeToken)  
  
    // 2  
    var zoneIds = [CKRecordZone.ID]()  
    operation.recordZoneWithIDChangedBlock = { zoneId in  
        zoneIds.append(zoneId)  
    }  
  
    // 3  
    operation.changeTokenUpdatedBlock = { [weak self] changeToken in  
        self?.privateDatabaseChangeToken = changeToken  
    }  
  
    // 4  
    operation.fetchDatabaseChangesCompletionBlock = { [weak self]  
changeToken, success, error in  
        self?.privateDatabaseChangeToken = changeToken  
  
        if zoneIds.count > 0 && error == nil {  
            self?.fetchZoneChangesInZones(zoneIds, completionHandler)  
        } else {  
            completionHandler(error)  
        }  
    }  
  
    privateDatabase.add(operation)  
}  
}
```

The previous method retrieves only the record changes from the CloudKit database in the following steps:

1. `CKFetchDatabaseChangesOperation` is created. This operation will retrieve changes in the database zones. The latest change token is passed to this operation to make sure it only returns data that has changed since the last successful fetch.
2. The fetch operation will not return all changes zones at once. Instead, every changed zone is provided one by one, so they should be stored in an array for easy access at a later time.

3. At certain times in the refresh cycle, the operation will receive a change token from CloudKit. A single fetch operation might result in more than one change token. This could occur if there are many changes to be fetched from the server. In that case, the server will return the changed data in batches, allowing you to process the changes as they come in. It's essential that you store the latest token you have received from the server.
4. Provide a closure to the operation that should be executed when the fetch operation completes. This method also receives a change token, so you should store this token locally since that token represents the completed fetch operation. If no errors occurred and at least one database zone has changed, a new method is called. This method is responsible for fetching all changes to the database records, and you will implement it next.

Once the server has informed the app about all of the changed zones in the database, the app should then ask CloudKit for the relevant record changes. A single operation can retrieve data across multiple zones so your app won't have to issue many requests to CloudKit if there are several zones that have changes. Add the following extension to `CloudStore.swift`:

```
extension CloudStore {  
    func fetchZoneChangesInZones(_ zones: [CKRecordZone.ID], _  
completionHandler: @escaping (Error?) -> Void) {  
    // 1  
    var fetchConfigurations = [CKRecordZone.ID:  
CKFetchRecordZoneChangesOperation.ZoneConfiguration]()  
  
    for zone in zones {  
        if let changeToken =  
UserDefaults.standard.zoneChangeToken(forZone: zone) {  
            let configuration =  
CKFetchRecordZoneChangesOperation.ZoneConfiguration(previousServerChan  
geToken: changeToken, resultsLimit: nil, desiredKeys: nil)  
            fetchConfigurations[zone] = configuration  
        }  
    }  
  
    let operation = CKFetchRecordZoneChangesOperation(recordZoneIDs:  
zones, configurationsByRecordZoneID: fetchConfigurations)  
  
    // 2  
    var changedMovies = [CKRecord]()  
    var changedFamilyMembers = [CKRecord]()  
    operation.recordChangedBlock = { record in  
        if record.recordType == "Movie" {  
            changedMovies.append(record)  
        }  
    }  
}
```

```
        } else if record.recordType == "FamilyMember" {
            changedFamilyMembers.append(record)
        }
    }

    operation.fetchRecordZoneChangesCompletionBlock = { [weak self]
error in
    for record in changedMovies {
        print(record["title"])
    }

    for record in changedFamilyMembers {
        print(record["name"])
    }

    completionHandler(error)
}

// 3
operation.recordZoneFetchCompletionBlock = { recordZone,
changeToken, data, moreComing, error in
    UserDefaults.standard.set(changeToken, forZone: recordZone)
}

privateDatabase.add(operation)
}
}
```

The preceding method is quite long but it follows a flow that is similar to the other two. Let's walk through this code step by step again:

1. When you fetch changes for record zones, each zone will have its own change token. This step uses another helper that was added to `UserDefault`s to conveniently store change tokens for each individual zone. The zones and their change tokens should be provided to `CKFetchRecordZoneChangesOperation` through a configuration dictionary. The operation itself is initialized with the dictionary and a list of zones that have changed.
2. The operation executes a callback for every changed record that it receives from the server. To make processing at a later time easier, the records for family members and movies are stored in their own lists. Note that both movies and family members are sent to the app as `CKRecord` instances. `CKRecord` has several specific CloudKit properties and methods, but otherwise, it behaves a lot like a dictionary. You will learn more about `CKRecord` instances later when you store your objects in Core Data.

3. `CKFetchRecordZoneChangesOperation` will call a closure for every zone that it processes. This callback receives a lot of arguments, such as the database zone that the closure was called for, the corresponding change token, and whether more data is coming in. The most interesting cases are the database zone and the change token. These two properties are used to store the latest change token for the corresponding zone in `UserDefault`s.

At this point, the `CloudStore` object contains all of the logic required to subscribe to database changes, retrieve them, and then print the objects that were fetched. To use `CloudStore`, you will need to update some code in `AppDelegate`.

Configuring your AppDelegate

As mentioned, silent push notifications work very similar to the way background fetch works. Just like background fetch, there is a special `AppDelegate` method that is called when a silent push notification arrives. You need to implement this method so you can handle the incoming notifications and fetch the change data from CloudKit. Add the following method to `AppDelegate`:

```
func application(_ application: UIApplication,  
didReceiveRemoteNotification userInfo: [AnyHashable : Any],  
fetchCompletionHandler completionHandler: @escaping  
(UIBackgroundFetchResult) -> Void) {  
  
    guard let dict = userInfo as? [String: NSObject] else {  
        completionHandler(.failed)  
        return  
    }  
  
    cloudStore.handleNotification(dict) { result in  
        completionHandler(result)  
    }  
}
```

This method is called by iOS when your app receives a silent push notification from CloudKit. This message doesn't do much apart from telling `CloudStore` to process the notification and then it calls `completionHandler` to inform iOS about the result of the fetch. As you can see, this is very similar to background fetch apart from the fact that this method is called when the CloudKit server sends a push message to the app instead of when iOS decides to do so.

Lastly, add the following code at the end of

```
application(_: didFinishLaunchingWithOptions:):  
  
    cloudStore.subscribeToChangesIfNeeded { [weak self] error in  
        DispatchQueue.main.async {  
            application.registerForRemoteNotifications()  
        }  
  
        if error == nil {  
            self?.cloudStore.fetchDatabaseChanges { fetchError in  
                if let error = fetchError {  
                    print(error)  
                }  
            }  
        }  
    }  
}
```

The preceding code takes care of telling `CloudStore` to subscribe to changes. It also registers the app for remote notifications so CloudKit can send silent push messages to the app. Note that this type of push messaging does not require you to explicitly ask the user to send them push notifications. This type of permission is only required if you want to send the user visual push notifications. You can read more about push notifications in [Chapter 22, Implementing Rich Notifications](#).

At this point, you can add a couple of records to your CloudKit database. Since you haven't implemented sending data to CloudKit yet, this is the best way for you to verify that your code works as expected. You should see the names of the movies and family members you have added in the CloudKit dashboard printed to the console. Once you have the app running, try adding a new family member while the app is open. You should see the name of the new family member appear in the console right after you add the record to the CloudKit dashboard. Pretty neat, right?



To use the private database in CloudKit, a user that is logged in to CloudKit on their device is required. Make sure to run your app on a physical device that has a logged-in iCloud user (preferably the same account you use as your Apple developer account) or log an iCloud user in on the simulator by navigating to the Settings app on the simulator.

Storing data in CloudKit

To store objects in CloudKit, you use `CKModifyRecordsOperation`. This operation takes a list of records that should be created, updated, or removed. Every record is connected to a certain zone, should have a unique identifier, and can have different properties associated with it. If you add a property to an object that doesn't exist in CloudKit yet, CloudKit will automatically update the data model on the server using the data you sent. You can even create entirely new models this way. You can add the following code to `CloudStore.swift` to store a very basic version of a family member's record:

```
extension CloudStore {
    func storeFamilyMember(_ familyMember: FamilyMember, _ completionHandler: @escaping (Error?) -> Void) {
        let defaultZoneId = CKRecordZone.ID(zoneName: "moviesZone",
                                              ownerName: CKCurrentUserDefaultName)
        let recordId = CKRecord.ID(recordName: UUID().uuidString, zoneID: defaultZoneId)
        let record = CKRecord(recordType: "FamilyMember", recordID: recordId)
        record["name"] = familyMember.name!

        let operation = CKModifyRecordsOperation(recordsToSave: [record],
                                                recordIDsToDelete: [])
        operation.modifyRecordsCompletionBlock = { records, recordIds, error in
            guard let records = records, error == nil else {
                completionHandler(error)
                return
            }

            print(records)
        }

        privateDatabase.add(operation)
    }
}
```

The preceding code creates a zone ID and a record ID that will be used to identify the record in the database. Next, a family member record is created and the family member name is added to it. Lastly, `CKModifyRecordsOperation` is created and added to the database to send the record to CloudKit. To call this method when a new family member is added, update the `saveFamilyMember(withName:)` method in `FamilyMembersViewController` as follows:

```
func saveFamilyMember(withName name: String) {
    let moc = persistentContainer.viewContext

    moc.persist {
        let familyMember = FamilyMember(context: moc)
        familyMember.name = name
        familyMember.uuidString = UUID().uuidString

        self.cloudStore.storeFamilyMember(familyMember) { _ in
            // no action
        }
    }
}
```

This snippet passes the family member to the cloud store so it can convert it to `CKRecord` and store it in CloudKit. Try adding a new family member in the app, you should see your newly-inserted record in the console in Xcode; when you refresh the list of family members in the CloudKit dashboard, a new entry should have appeared.

This implementation of adding a family member to CloudKit is too basic because there is no way to associate one of the local family members to the family members stored in CloudKit. The next section will implement a proper local cache of the data in CloudKit by adapting the existing Core Data model a little bit so you can import data from the CloudKit server, send new family members and movies to CloudKit, and update existing family members when they add new movies to their favorites.



Since your app stores database change tokens in `UserDefault`s, you should delete `MustC` from your device to remove the existing change tokens. This allows you to start fresh and receive all data again.

Combining CloudKit and Core Data

When you implement CloudKit in your app, it is always highly recommended to have a local cache of all CloudKit data. The previous section showed you how to fetch changes from the CloudKit database and how to process them. You also saw a very basic sample of how to send data to CloudKit. In this section, you will implement some helper methods on the `Movie` and `FamilyMember` classes. You will also add some extra properties to the Core Data models to facilitate CloudKit, and finally, you will implement the importing and exporting of CloudKit data.

Preparing the Core Data models for CloudKit

You might have noticed the following line in the code for creating a new `CKRecord` that was shown in the previous section:

```
let recordId = CKRecord.ID(recordName: UUID().uuidString, zoneID:  
    defaultZoneId)
```

The `recordName` that is set on `CKRecord.ID` is the unique identifier that CloudKit uses to store records. When you want to import data from CloudKit, you can use `recordName` to check whether you have already saved a record in your local database. In addition to this unique identifier, CloudKit itself stores metadata about records. You saw this metadata when you added a new record type in the CloudKit dashboard and all these new properties that you didn't add yourself showed up.

Apart from the unique identifier, none of the metadata is very relevant to use in the `MustC` app, so adding all fields to the Core Data model would be a waste. Luckily, `CKRecord` has a convenient method that allows you to encode all of the automatically-added metadata into a `Data` object that you can easily store in Core Data. This means that you'll need to add two new properties to both the `FamilyMember` and `Movie` models:

- `recordName (String)`
- `cloudKitData (Data)`

Go ahead and add these two properties to your models in the Core Data model editor.

After adding the `recordName` and `cloudKitData` attributes, you should add a couple of helpers to your models. Create a new file in the Models folder and name it `FamilyMember.swift`. Add the following extension to this file:

```
extension FamilyMember {
    // 1
    func recordIDForZone(_ zone: CKRecordZone.ID) -> CKRecord.ID {
        return CKRecord.ID(recordName: self.recordName!, zoneID: zone)
    }

    // 2
    func recordForZone(_ zone: CKRecordZone.ID) -> CKRecord {
        let record: CKRecord

        // 3
        if let data = cloudKitData, let coder = try?
            NSKeyedUnarchiver(forReadingFrom: data) {
            coder.requiresSecureCoding = true
            record = CKRecord(coder: coder)!
        } else {
            record = CKRecord(recordType: "FamilyMember", recordID:
                recordIDForZone(zone))
        }

        record["name"] = name!

        // 4
        if let movies = self.movies as? Set<Movie> {
            let references: [CKRecord.Reference] = movies.map { movie in
                let movieRecord = movie.recordForZone(zone)
                return CKRecord.Reference(record: movieRecord, action: .none)
            }

            record["movies"] = references
        }

        return record
    }

    // 5
    static func find(byIdentifier recordName: String, in moc:
        NSManagedObjectContext) -> FamilyMember? {
        let predicate = NSPredicate(format: "recordName == %@", recordName)
        let request: NSFetchedResultsController<FamilyMember> =
            FamilyMember.fetchRequest()
        request.predicate = predicate
    }
}
```

```
        guard let result = try? moc.fetch(request)
            else { return nil }

        return result.first
    }
}
```

A lot is going on in this snippet, so let's go over it step by step:

1. The first comment highlights a convenient helper method that creates a record ID. This helper method receives an existing zone ID and uses the new `recordName` property that you just added in the Core Data model editor.
2. This method is used to convert the `FamilyMember` model into a `CKRecord` instance. This method will be used when you send data to CloudKit.
3. `CKRecord` objects can be created in different ways. One way is through the `CKRecord(recordType:recordID:)` initializer. You can also use an instance of `NSCoder` to create a `CKRecord` with the `CKRecord(coder:)` initializer. An `NSCoder` can convert objects into data and vice versa. So, in this case, a special version of `NSCoder`, called `NSKeyedUnarchiver`, is used to convert the metadata that is stored as `Data` in Core Data back into a `CKRecord` instance. If the Core Data object has just been added, it won't have any CloudKit metadata yet, so a new `CKRecord` instance should be created.
4. To make sure all movies for the family member are sent to CloudKit, it is required to loop over each movie and create a `CKRecord`. Reference to the object. This list of references is then assigned to the movie record.
5. To import CloudKit objects, the code must be able to look up existing movie records using the record's `recordName`. This method is used to look up family members by their record name.

The `Movie` object should receive similar helper methods to the ones you just added to `FamilyMember`. Add the following methods to the extension in `Movie.swift`:

```
func recordIDForZone(_ zone: CKRecordZone.ID) -> CKRecord.ID {
    return CKRecord.ID(recordName: self.recordName!, zoneID: zone)
}

func recordForZone(_ zone: CKRecordZone.ID) -> CKRecord {
    let record: CKRecord

    if let data = cloudKitData, let coder = try?
        NSKeyedUnarchiver(forReadingFrom: data) {

```

```
        coder.requiresSecureCoding = true
        record = CKRecord(coder: coder) !
    } else {
        record = CKRecord(recordType: "Movie", recordID:
recordIDForZone(zone))
    }

    record["title"] = title!
    record["rating"] = popularity
    record["remoteId"] = remoteId

    return record
}

static func find(byIdentifier recordName: String, in moc:
NSManagedObjectContext) -> Movie? {
    let predicate = NSPredicate(format: "recordName == %@", recordName)
    let request: NSFetchedRequest<Movie> = Movie.fetchRequest()
    request.predicate = predicate

    guard let result = try? moc.fetch(request)
        else { return nil }

    return result.first
}

static func find(byIdentifiers recordNames: [String], in moc:
NSManagedObjectContext) -> [Movie] {
    let predicate = NSPredicate(format: "ANY recordName IN %@", recordNames)
    let request: NSFetchedRequest<Movie> = Movie.fetchRequest()
    request.predicate = predicate

    guard let result = try? moc.fetch(request)
        else { return [] }

    return result
}
```

The code in the preceding snippet should speak for itself. One interesting addition is the `find(byIdentifiers:)` method. Instead of taking just a single record name, this method takes a list of record names. When you import family members from CloudKit, a single family member could have multiple movies in their favorites. Instead of fetching each movie individually, this method allows you to retrieve all matching movies at once.

Your Core Data models are now fully compatible with CloudKit, and you're ready to write the code that will import the data from the CloudKit servers and add them to your local Core Data database.

Importing CloudKit data

You have already written most of the code to retrieve changed records from the CloudKit database. The next step is to implement the code that takes the `CKRecord` instances that you receive from CloudKit and convert them into the correct Core Data models. Before you implement the import methods,

`update fetchRecordZoneChangesCompletionBlock in
fetchZoneChanges(_:_:) on CloudStore` as follows:

```
let backgroundContext = persistentContainer.newBackgroundContext()

operation.fetchRecordZoneChangesCompletionBlock = { [weak self] error
in
    for record in changedMovies {
        self?.importMovie(withRecord: record, withContext:
backgroundContext)
    }

    for record in changedFamilyMembers {
        self?.importFamilyMember(withRecord: record, withContext:
backgroundContext)
    }

    completionHandler(error)
}
```

The preceding snippet uses `persistentContainer` to obtain a background managed object context. This type of managed object context runs off the main thread, making it ideal for operations that could take a while. Since you can't ever be sure about the amount of data that CloudKit sends your way, it's a good idea to perform all import work on a background managed object context. When using background contexts, it's important that you pay extra attention to the Core Data objects you're working with. You should always make sure to create and use Core Data objects on a single thread. You can't create an object in one managed object context and then save it in the other. If you do accidentally mix up threads, you could end up with some very difficult-to-debug crashes.

Inside of the completion block, two new methods are called: one that imports movies and one that imports family members. The reason they are called in this order is to make sure all movies are imported first and the family members second. This ensures that all the family member's favorite movies exist in the database so they can be associated with the family member immediately.

Add the following extension to `CloudStore.swift` to implement the importers:

```
extension CloudStore {
    func importMovie(withRecord record: CKRecord, withContext moc: NSManagedObjectContext) {
        moc.persist {
            let identifier = record.recordID.recordName
            let movie = Movie.find(byIdentifier: identifier, in: moc) ??
            Movie(context: moc)
            movie.title = record["title"] ?? "unkown-title"
            movie.remoteId = record["remoteId"] ?? 0
            movie.popularity = record["rating"] ?? 0.0
            movie.cloudKitData = record.encodedSystemFields
            movie.recordName = identifier
        }
    }

    func importFamilyMember(withRecord record: CKRecord, withContext moc: NSManagedObjectContext) {
        moc.persist {
            let identifier = record.recordID.recordName
            let familyMember = FamilyMember.find(byIdentifier: identifier,
            in: moc) ?? FamilyMember(context: moc)
            familyMember.name = record["name"] ?? "unkown-name"
            familyMember.cloudKitData = record.encodedSystemFields
            familyMember.recordName = identifier

            if let movieReferences = record["movies"] as?
            [CKRecord.Reference] {
                let movieIds = movieReferences.map { reference in
                    return reference.recordID.recordName
                }

                familyMember.movies = NSSet(array: Movie.find(byIdentifiers:
                movieIds, in: moc))
            }
        }
    }
}
```

The first method in this extension imports movie objects. If you look at this method carefully, you'll find that it's relatively straightforward. The record's unique identifier is used to look up a movie in Core Data. If no movie could be found, a new one is created. All the following steps extract data from `CKRecord` and apply it to the `Movie` instance. Since the all code is wrapped in a `moc.persist` closure, the new movie will be saved immediately.

The method to import family members is slightly more complex. The basics are the same as they are for the movie import. However, the family member importer has to extract the references to movie records and then map the references to record names. These record names are then used to look up all movies in the list of favorites for this family member, and then assigns this list to the family member.

Try deleting and reinstalling the app now. You should automatically see the list of family members appear.



Does the list of family members only appear for you if you start the app for the second time? Your `viewContext` might be misconfigured. Make sure that you set `container.viewContext.automaticallyMergesChangesFromParent` to `true` in the `lazy` variable that creates the persistent container in `AppDelegate`.

Just one more step until you have a complete CloudKit-enabled application that sends Core Data models to CloudKit.

Sending Core Data models to CloudKit

To send Core Data models to CloudKit, you must convert them to `CKRecord` instances. So far, you've learned how to create an instance of `CKRecord` and you even implemented a helper method on the `Movie` and `FamilyMember` objects. Take another look at `recordForZone(_:_)` on both objects to make sure you understand how the models are converted to records. When you convert Core Data objects to CloudKit models, it's important that you properly use the CloudKit metadata. If you attempt to store an existing object with the correct record name, but you omit the metadata, CloudKit won't be able to save your object because it can't accurately compare the version that it currently has stored and the new version you are trying to send it.

Before you update the `storeFamilyMember(_:_)` method, update the `saveMovie(withName:)` method in `MoviesViewController` so family members get stored when you add a new family member or when you update one by assigning it a new movie. Update the method as follows:

```
func saveMovie(withName name: String) {
    // ...

    let helper = MovieDBHelper()
    helper.fetchRating(forMovie: name) { remoteId, rating in
        guard let rating = rating,
              let remoteId = remoteId
        else { return }

        moc.persist {
            movie.popularity = rating
            movie.remoteId = Int64(remoteId)

            self.cloudStore.storeFamilyMember(familyMember) { _ in
                // no action
            }
        }
    }
    // ...
}
```

The only thing that has changed in this method is that the family member is now stored after the movie's popularity rating is fetched. The next and final step is to implement the new version of `storeFamilyMember(_:_)`, as follows:

```
func storeFamilyMember(_ familyMember: FamilyMember, _ completionHandler: @escaping (Error?) -> Void) {
    // 1
    guard let movies = familyMember.movies as? Set<Movie> else {
        completionHandler(nil)
        return
    }

    let defaultZoneId = CKRecordZone.ID(zoneName: "moviesZone",
                                         ownerName: CKCurrentUserDefaultName)

    // 2
    var recordsToSend = movies.map { movie in
        movie.recordForZone(defaultZoneId)
    }
    recordsToSend.append(familyMember.recordForZone(defaultZoneId))

    let operation = CKModifyRecordsOperation(recordsToSend:
```

```
recordsToSave, recordIDsToDelete: nil)

operation.modifyRecordsCompletionBlock = { records, recordIDs, error
in

    guard let records = records, error == nil else {
        completionHandler(error)
        return
    }

    // 3
    for record in records {
        if record.recordType == "FamilyMember" {
            familyMember.managedObjectContext?.persist {
                familyMember.cloudKitData = record.encodedSystemFields
            }
        } else if record.recordType == "Movie",
            let movie = movies.first(where: { $0.recordName ==
record.recordID.recordName }) {

            familyMember.managedObjectContext?.persist {
                movie.cloudKitData = record.encodedSystemFields
            }
        }
    }

    completionHandler(error)
}

privateDatabase.add(operation)
}
```

Even though the preceding method is quite long, it should be relatively straightforward; there are some parts that are worth taking a closer look at:

1. Because the `movies` property on `FamilyMember` is an `NSSet`, it needs to be converted to a `Set<Movie>` so the movies can be looped over. This conversion should never fail, so if it does, something is wrong and it makes no sense to continue saving the family member to CloudKit.
2. Extract all movie records from the family member to obtain a list of records that should be saved. The family member itself is then also added to this list, so it's also saved to CloudKit.

- Once the save operation is created, and all records are saved, the CloudKit metadata is added to their corresponding Core Data objects, and the objects are then saved. Note that the managed object context is obtained by calling `familyMember.managedObjectContext`. Doing this ensures that the family member and its movies are updated in the correct managed object context.

If you run your app now, you should be able to import records from CloudKit automatically. When you add records to CloudKit while the app is running, it will automatically update the user interface. And when you add new family members and movies to the Core Data database, they are automatically saved to CloudKit.



If you want to test your app on multiple devices at once, you can use different simulators that are signed in to the same iCloud account. When you add a family member on one simulator, it will automatically appear on all other simulators that have the same iCloud user.

Summary

In this chapter, you learned a lot about CloudKit. You learned what CloudKit is and what it's used for. You added the required capabilities to your app, and then you explored your new CloudKit container in the CloudKit dashboard. You used the dashboard to define some models that you will later use in your app.

After configuring the CloudKit dashboard, you got your hands dirty by implementing a CloudKit sync feature for the **MustC** app. You learned that CloudKit makes extensive use of `Operations` to manage database operations. You also had to set up `AppDelegate` so it would subscribe the app to changes in the user's database. Then you learned how you can fetch updates from CloudKit using a change token, and how to send new records to CloudKit.

To wrap the feature up, you adapted your Core Data store to make it act as a local cache for CloudKit. You saw how to convert a Core Data model object to a CloudKit record and vice versa. This chapter made some extensive use of extensions on objects, such as `UserDefault`s to make working with them more convenient. It's always a good idea to extract code you don't want to write too often into an extension if possible.

This chapter wraps up the **MustC** app for now. In the next chapters, you will dive into some of the excellent frameworks iOS has to offer, such as ARKit, Core Location, and SiriKit.

Questions

1. How does the CloudKit server notify apps about updates?
 - a) Background Fetch.
 - b) Silent push notifications.
 - c) It doesn't. Apps should ask for changes on launch.
2. Which object do you send along with a fetch request to CloudKit so you only receive new changes?
 - a) A change token.
 - b) A timestamp.
 - c) The last object you received updates for.
3. How does your code know whether a certain `CKRecord` is a movie, family member, or something else?
 - a) Through `recordName`.
 - b) By checking what properties are available.
 - c) Through `recordType`.
4. What kind of data do you have to store alongside your objects in Core Data to be able to properly sync local data with CloudKit?
 - a) Their change tokens.
 - b) The encoded CloudKit metadata.
 - c) The zone they were retrieved from.

5. What's a good place to store server change tokens?
 - a) Core Data.
 - b) In an array.
 - c) UserDefaults.
6. Why is it smart to import CloudKit data on a background managed object context?
 - a) Because you don't know how long the import will take and you don't want to block the main thread.
 - b) Because background contexts are faster at writing data.
 - c) To prevent simultaneous read and write operations.

Further reading

- Refer to Chapter 14, *Being Proactive with Background Fetch*, for more information about background fetch features
- Refer to Chapter 22, *Implementing Rich Notifications*, for more information about push notifications
- Refer to Chapter 28, *Offloading Tasks with Operations and GCD*, to learn more about GCD and threading

16

Using Augmented Reality

One of the major features that Apple shipped as part of iOS 11 was **ARKit**. ARKit enables developers to create amazing Augmented Reality experiences with only a minimal amount of code. Apple has continuously worked on improving ARKit, resulting in the release of ARKit 2 at WWDC 2018.

In this chapter, you will learn what ARKit is, how it works, what you can do with it, and how you can implement an Augmented Reality Art gallery that uses several of ARKit 2 new features, such as image tracking. This chapter covers the following ARKit topics:

- Understanding ARKit
- Using ARKit Quicklook
- Exploring SpriteKit
- Exploring SceneKit
- Implementing an Augmented Reality gallery

By the end of this chapter, you will be able to integrate ARKit in your apps and implement your own ARKit experiences.

Understanding ARKit

Augmented Reality (AR) is a topic that has captured the interest of app developers and designers for a long time now. Implementing an excellent AR experience had not been easy though, and many applications haven't lived up to the hype. Small details such as lighting, and detecting walls, floors, and other objects have always been extremely complicated to implement and getting these details wrong has a negative impact on the quality of an AR experience.

Augmented reality apps usually have at least some of the following features:

- They show a camera view.
- Content is shown as an overlay on the camera.
- Content responds appropriately to device movement.
- Content is attached to a specific location in the world.

Even though this list of features is simple, they aren't all trivial to implement. An AR experience relies heavily on reading the motion sensors from the device, as well as using image analysis to determine exactly how a user is moving and to learn what a 3D map of the world should look like.

ARKit is Apple's way of giving developers the power to create great AR experiences. ARKit takes care of all the motion and image analysis to make sure you can focus on designing and implementing great content rather than getting slowed down by the intricate details involved in building an AR app.

Unfortunately, ARKit comes with a hefty hardware requirement for the devices that can run ARKit apps. Only devices with Apple's A9 chip or newer can run ARKit. This means that any device older than the iPhone 6s or first iPad Pro cannot run ARKit apps.

Understanding how ARKit renders content

ARKit itself only takes care of the massive calculations related to keeping track of the physical world the user is in. To render content in an ARKit app, you must use one of the following three rendering methods:

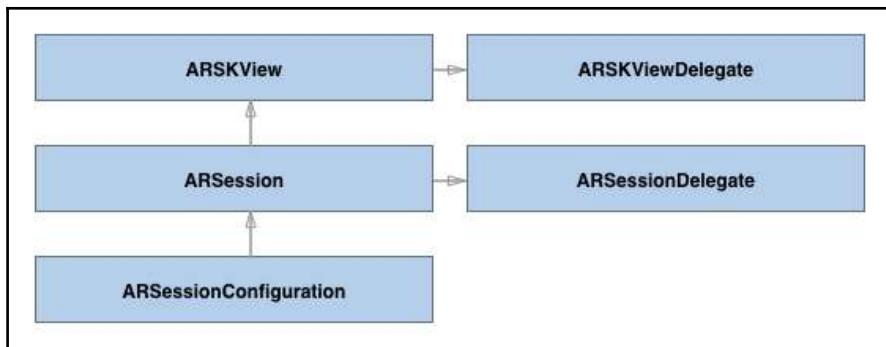
- SpriteKit
- SceneKit
- Metal

Later in this chapter, you will have a quick look at SpriteKit and SceneKit, and you will ultimately implement your Augmented Reality gallery using SceneKit. If you already have experience with any of the available rendering techniques, you should feel right at home when using ARKit.

Implementing ARKit in your app is not limited to manually rendering the contents you want to show in AR. In iOS 12, Apple has added a feature called **ARKit Quicklook**. You can implement a special view controller in your app that takes care of placing a 3D model you supply in a scene. This is ideal if you're implementing a feature that allows users to preview products or other objects in the real world.

Understanding how ARKit tracks the physical environment

To understand how ARKit renders content, it's essential that you understand how ARKit makes sense of the physical environment a user is in. When you implement an AR experience, you use an ARKit session. An ARKit session is represented by an instance of `ARSession`. Every `ARSession` uses an instance of `ARSessionConfiguration` to describe the tracking that it should do on the environment. The following diagram depicts the relationship between all objects involved in an ARKit session:



The preceding image shows how the session configuration is passed to the session. The session is then passed to a view that is responsible for rendering the scene. If you use SpriteKit to render the scene, the view is an instance of `ARSKView`. When you use SceneKit, this would be an instance of `ARSCNView`. Both the view and session have a delegate that will be informed about certain events that can occur during an ARKit session. You will learn more about these delegates later when you implement your AR gallery.

There are several different tracking options that you can configure on a session. One of the most basic tracking configurations is `AROrientationTrackingConfiguration`. This configuration only tracks the device's orientation, so not the user's movement in the environment. This kind of tracking monitors the device using three degrees of freedom. To be more specific, this tracking tracks the device's x, y, and z orientation. This kind of tracking is perfect if you're implementing something such as a 3D video where the user's movements can be ignored.

A more complex tracking configuration is `ARWorldTrackingConfiguration`, also known as **World tracking**. This type of configuration tracks the user's movements as well as the device's orientation. This means that a user can walk around an AR object to see it from all different sides. World tracking uses the device's motion sensors to determine the user's movements and the device orientation. This is very accurate for short and small movements, but not accurate enough to track movements over long periods of time and distances. To make sure the AR experience remains as precise as possible, world tracking also performs some advanced computer vision tasks to analyze the camera feed to determine the user's location in an environment.

In addition to tracking the user's movements, world tracking also uses computer vision to make sense of the environment that the AR session exists in. By detecting certain points of interest in the camera feed, world tracking can compare and analyze the position of these points in relation to the user's motion to determine the distances and sizes of objects. This technique also allows world tracking to detect walls and floors for instance.

The world tracking configuration stores everything it learns about the environment in an `ARWorldMap`. This map contains all `ARAnchor` instances that represent different objects and points of interest that exist in the session.

There are several other special tracking types that you can use in your app. For instance, you can use `ARFaceTrackingConfiguration` on devices with a TrueDepth camera to track a user's face. This kind of tracking is perfect if you want to recreate Apple's Animoji feature that was added to the iPhone X and newer in iOS 12.

You can also configure your session, so it automatically detects certain objects or images in a scene. To implement this, you can use `ARObjectScanningConfiguration` to scan for specific items or `ARIImageTrackingConfiguration` to identify still images.

Before you get your hands dirty with implementing an ARKit session, let's explore the new ARKit Quicklook session to see how simple it is for you to allow users of your app to preview items in AR.

Using ARKit Quicklook

One of the great benefits that AR brings to end users is that it is now possible to preview certain objects in the real world. For instance, when you buy a new sofa, you might want to see what it looks like in the real world. Of course, it was possible to implement features such as this in iOS 11 using ARKit, and many developers have, but it wasn't as easy as it could be.

iOS users can preview content using a feature called Quicklook. **Quicklook** can be used to preview certain types of content without having to launch any specific applications. This is convenient for users because they can quickly determine whether a particular document is the document they are looking for by previewing it in Quicklook.

In iOS 12, Apple added the USDZ file format to the content types that can be previewed using Quicklook. Apple's USDZ format is a 3D file format based on Pixar's USD format that is used to represent 3D objects. Using Quicklook for 3D models is not just available in apps, ARKit Quicklook can also be integrated on the web. Developers can use a special HTML tag on their web pages to link to a USDZ and Safari will display the model in an ARKit Quicklook view controller.

Before you implement your AR gallery, it's a good idea to get a feeling for how AR works on iOS by implementing the ARKit Quicklook view controller to show one of the models that Apple provides on <https://developer.apple.com/arkit/gallery/>. To download a model you like, all you need to do is navigate to this page on your Mac and click on an image. The USDZ file should start downloading automatically.

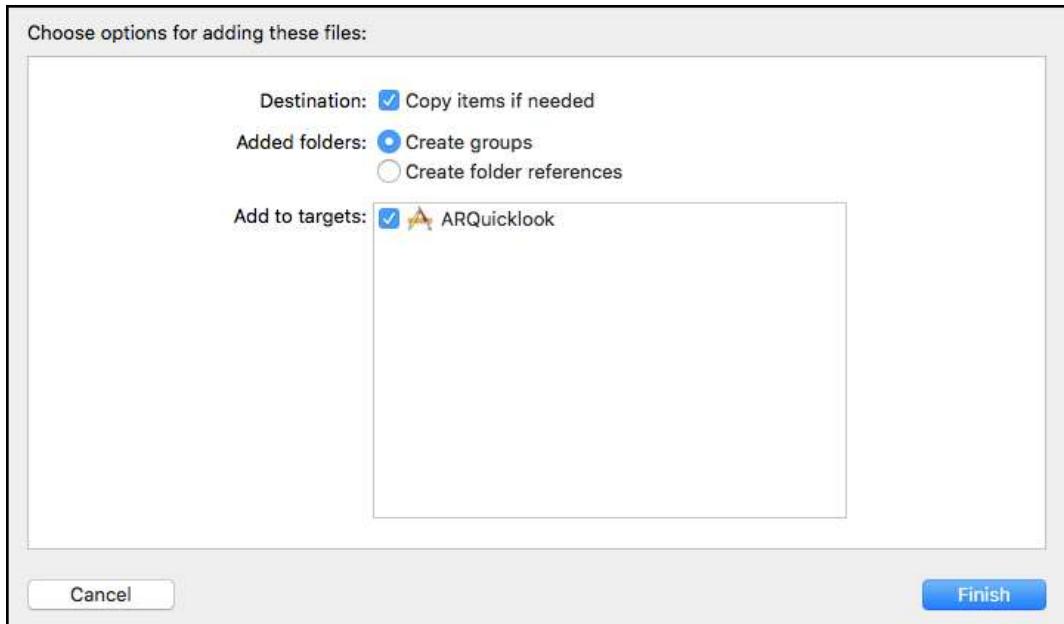


Navigate to the ARKit gallery on a device that supports ARKit and tap on one of the models to see what ARKit Quicklook in Safari looks like.

Implementing the ARKit Quicklook view controller

After obtaining a USDZ file from Apple's gallery, also make sure to capture the image that belongs to this file. Taking a screenshot of the model should be fine for testing purposes. Make sure to prepare your image in the different required sizes by scaling your screenshot up to two and three times the size of your screenshot.

Create a new project in Xcode and pick a name for your project. The sample project in this book's code bundle is called **ARQuickLook**. Add your prepared image to the `Assets.xcassets` file. Also, drag your `USDZ` file into Xcode and make sure to add it to the app target by checking your app's checkbox when importing the file:



Next, open the storyboard file and drag an image view to the view controller. Add the proper constraints to the image, so it's centered in the view controller and give it a width and height of 200 points. Make sure to check the **User Interaction Enabled** checkbox in the **Attributes Inspector** and set your model image as the image for the image view.

After doing this, open `ViewController.swift`, add `@IBOutlet` for the image view, and connect the image in the storyboard to this outlet. If the details regarding outlets are a little bit fuzzy right now, refer to the sample project in the code bundle for a refresher. The image view in the sample project uses an outlet called `guitarImage`.

The next steps to implement Quicklook for the USDZ model are to add a tap gesture recognizer to the image view and then trigger the Quicklook view controller when a user taps on the image.

Quicklook uses delegation to object one or more items that it should preview from a data source. It also uses a delegate to obtain the source view from which the Quicklook preview should animate. This flow applies to all different kinds of files that you can preview using Quicklook.

To begin implementing Quicklook, you must import the `QuickLook` framework. Add the following import statement to the top of `ViewController.swift`:

```
import Quicklook
```

Next, set up the tap-gesture recognizer for the image by adding the following code to `viewDidLoad()`:

```
let tapGesture = UITapGestureRecognizer(target: self, action:  
#selector(presentQuicklook))  
guitarImage.addGestureRecognizer(tapGesture)
```

The next step is to implement `presentQuicklook()`. This method will create a Quicklook view controller, set the delegate and data source, and then present the Quicklook view controller to the user. Add the following implementation for this method to the `ViewController` class:

```
@objc func presentQuicklook() {  
    let previewViewController = QLPreviewController()  
    previewViewController.dataSource = self  
    previewViewController.delegate = self  
  
    present(previewViewController, animated: true, completion: nil)  
}
```

This implementation should not contain any surprises for you. `QLPreviewController` is a `UIViewController` subclass that is responsible for displaying the content it receives from its data source. It is presented in the same way you would present any other view controller, by calling `present(_:animated:completion:)`.

The final step is to implement the data source and delegates. Add the following extensions to `ViewController.swift`:

```
extension ViewController: QLPreviewControllerDelegate {
    func previewController(_ controller: QLPreviewController,
    transitionViewFor item: QLPreviewItem) -> UIView? {
        return guitarImage
    }
}

extension ViewController: QLPreviewControllerDataSource {
    func numberOfPreviewItems(in controller: QLPreviewController) -> Int
    {
        return 1
    }

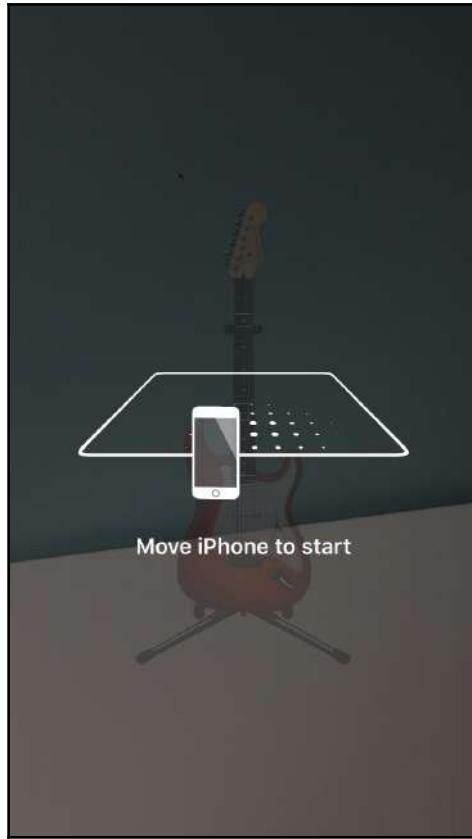
    func previewController(_ controller: QLPreviewController,
    previewItemAt index: Int) -> QLPreviewItem {
        let fileUrl = Bundle.main.url(forResource: "stratocaster",
    withExtension: "usdz")!
        return fileUrl as QLPreviewItem
    }
}
```

The first extension you added makes `ViewController` conform to `QLPreviewControllerDelegate`. When the preview controller is about to present the 3D model, it wants to know which view is the source for the transition that is about to happen. It's recommended to return the view that acts as a preview for the Quicklook action from this method. In this case, the preview is the image of the 3D model.

The second extension acts as the Quicklook data source. When you implement Quicklook for ARKit, you can only return a single item. So when the preview controller asks for the number of items in the preview, you should always return 1. The second method in the data source provides the item that should be previewed in the preview controller. All you need to do here is obtain the file URL for the item you wish to preview. In the sample app, the Stratocaster model from Apple's gallery is used. If your model has a different name, make sure to use the correct filename.

After obtaining the URL that points to the image in the app bundle, it should be returned to the preview controller as a `QLPreviewItem`. Luckily, `URL` instances can be converted to `QLPreviewItem` instances automatically.

If you run your app now, you can tap on your image of the 3D model to begin previewing it. You can preview the image on its own, or you can choose to preview it in AR. If you tap this option, the preview controller will tell you to move your device around:



To make a mapping of the world around you, ARKit requires some samples of the environment. When you move your device around, make sure not just to tilt it, but physically move it. Doing this will help ARKit discover trackable features in your surroundings.

Once ARKit has enough data about your surroundings, you can place the 3D model in the environment, scale it by pinching, rotate it, and move it around in the space. Note that the model is placed on a flat surface such as a table or the floor automatically rather than awkwardly floating around.

Also note that ARKit applies very realistic lighting to your object. The visual data that ARKit gathers about the environment is used to create a lighting map that is applied to the 3D model to make it properly blend in with the context in which the object was placed:



While playing around with ARKit like this is a lot of fun, it's even more fun to create your own AR experiences. Since ARKit supports several rendering techniques, such as SpriteKit and SceneKit, the next two sections will spend a little bit of time explaining the very basics of SpriteKit and SceneKit. You won't learn how to build complete games or worlds with these frameworks. Instead, you will learn just enough to get you started with implementing either rendering engine in an ARKit app.

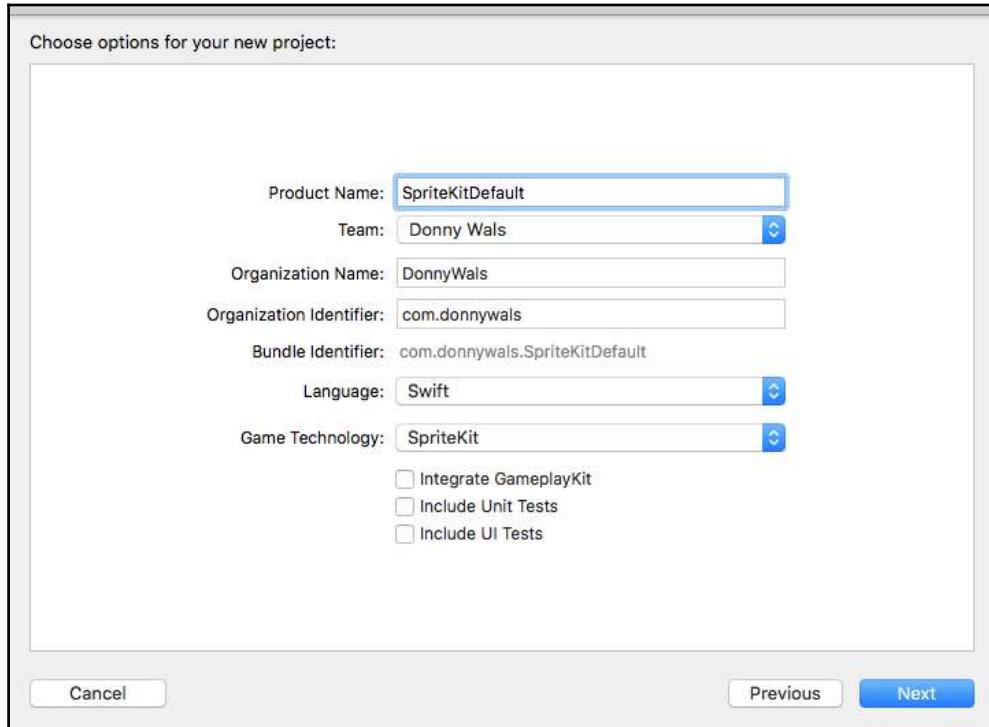
Exploring SpriteKit

SpriteKit is mostly used by developers to build two-dimensional games. SpriteKit has been around for quite some time already, and it has helped developers to create many successful games over the years. SpriteKit contains a full-blown physics simulation engine, and it can render many sprites at a time. A **sprite** represents a graphic in a game. A sprite could be an image for the player, but also a coin, an enemy, or even the floor that a player walks on. When sprites are mentioned in the context of SpriteKit, it is meant to refer to one of the nodes that are visible on the screen.

Because SpriteKit has a built-in physics engine, it can detect collisions between objects, apply forces to them, and more. This is pretty similar to what UIKit Dynamics is capable of.

To render content, SpriteKit uses scenes. These scenes can be considered levels or major building parts of a game. In the context of AR, you will find that you typically only need a single scene. A SpriteKit scene is responsible for updating the position and state of the scene. As a developer, you can hook into the rendering of frames through the `update(_:)` method of `SKScene`. This method is called every time SpriteKit is about to render a new frame for your game or ARKit scene. It is essential that this method's execution time is as short as possible, as a slow implementation of the `update(_:)` method will cause frames to drop, which is considered bad. You should always aim to maintain a steady 60 frames per second. This means that the `update(_:)` method should always perform its work in less than 1/60th of a second.

To begin exploring SpriteKit, create a new project in XCode and choose the **Game** template. Pick SpriteKit as the underlying game technology and give the project a name. For instance, **SpriteKitDefault**, as shown in the following screenshot:



When Xcode generates this project for you, you should notice some new files that you haven't seen before:

- GameScene.sks
- Actions.sks

These two files are to SpriteKit games what storyboards are to regular apps. You can use these to set up all nodes for your game scene or to set up reusable actions that you can attach to your nodes. We will not get into these files for now as they are pretty specific to game development.



If you are interested in learning all the ins and outs of SpriteKit, go ahead pick up **Swift 3 Game Development - Second Edition, by Stephen Haney**. This book goes in-depth about game development on iOS with SpriteKit.

If you build and run the sample project that Xcode provides, you can tap the screen to make new sprite nodes appear on the screen. Each node performs a little animation before it disappears. This isn't very special in itself, but it does contain a lot of valuable information. For instance, it shows you how to add something to a scene and how to animate it. Let's see exactly how this project is set up so you can apply this knowledge in case you wish to build an AR experience with SpriteKit at some point.

Creating a SpriteKit scene

SpriteKit games use a special type of view to render its contents. This special view is always an instance or subclass of `SKView`. If you want to use SpriteKit with ARKit, you should use `ARSKView` instead because that view implements some special AR-related behavior, such as rendering the camera feed.

The view itself usually doesn't do much work regarding managing the games or its child views. Instead, the `SKScene` that contains the view is responsible for doing this work. This is similar to how you usually work with view controllers in other apps.

When you have created a scene, you can tell an `SKView` to present the scene. From this moment on, your game is running. In the sample code for the game project you created earlier, the following lines take care of loading and presenting the scene:

```
if let scene = SKScene(fileNamed: "GameScene") {  
    scene.scaleMode = .aspectFill  
    view.presentScene(scene)  
}
```

When you create your scenes, you can choose whether you want to use `.sks` files or create scenes programmatically.

When you open the `GameScene.swift` file that Xcode created for you, most of the code should be pretty self-explanatory. When the scene is added to a view, a couple of `SKNode` instances are created and configured. The most interesting lines of code in this file are the following:

```
spinnyNode.run(SKAction.repeatForever(SKAction.rotate(byAngle:  
CGFloat(Double.pi), duration: 1)))  
spinnyNode.run(SKAction.sequence([SKAction.wait(forDuration: 0.5),  
SKAction.fadeOut(withDuration: 0.5), SKAction.removeFromParent()]))
```

These lines set up an animation sequence for the spinning squares that get added when you tap the screen. In SpriteKit, actions are the preferred way to set up animations. You can group, chain, and combine actions to achieve pretty complicated effects. This is one of the many powerful tools that SpriteKit has to offer.

If you examine the code a little bit more, you'll find that copies of `spinnyNode` are created every time the user taps on the screen, moves their finger, or lifts their finger. Each interaction produces a slightly different copy of `spinnyNode` so you can determine why `spinnyNode` was added to the scene by looking at its appearance.

Study this code, play around with it, and try to make sure that you grasp what it does. You don't have to become a SpriteKit expert by any means. Let's have a look at how SceneKit works to prepare to implement your Augmented Reality gallery.

Exploring SceneKit

When you're looking for a game framework that has excellent support for 3D games, SceneKit is a great candidate. SceneKit is Apple's framework for creating 3D games, and it is structured very similarly to how SpriteKit is set up.

Of course, SceneKit is entirely different from SpriteKit because it's used for 3D games rather than 2D games. Because of this, SceneKit also has very different ways of creating views and positioning them on screen. For instance, when you want to create a simple object and place it on the screen, you will see terms such as geometry and materials. These terms should be familiar to game programmers, but if you're an AR enthusiast, you will probably have to get used to the terminology.

This section will walk you through setting up a straightforward SceneKit scene that closely resembles a part of the AR gallery you will implement later. This should provide you with enough information to begin experimenting with SceneKit.



If you're left longing for more SceneKit knowledge by the end of this chapter, make sure to have a look at **3D iOS Games by Tutorials written by Chris Langue**. That book goes in-depth regarding 3D game development using SceneKit.

Creating a basic SceneKit scene

To practice your SceneKit knowledge, create a new project and instead of choosing the **Game** template, pick the **Single View Application** template. Of course, you are free to explore the default project Xcode creates for you when you choose the **Game** template with SceneKit, but it's not terribly useful for the AR gallery.

After creating your project, open the main storyboard and look for a **SceneKit view**. Drag this view into the view controller. You should notice that the view you just added to the view controller has replaced the default view entirely. Because of this, the `view` property on `ViewController` will not be a regular `UIView`, it will be an instance of `SCNView` instead. This is the view that will be used to render the SceneKit scene in.

Add the following code to `viewDidLoad()` in `ViewController.swift` to cast the `view` property from a `UIView` to an `SCNView`:

```
guard let sceneView = self.view as? SCNView  
else { return }
```

Similar to how SpriteKit works, SceneKit uses a scene to render its nodes in. Create an instance of `SCNScene` right after `guard` in `viewDidLoad()`, as shown:

```
let scene = SCNScene()  
sceneView.scene = scene  
sceneView.allowsCameraControl = true  
sceneView.showsStatistics = true  
sceneView.backgroundColor = UIColor.black
```

The preceding code creates a simple scene that will be used render all elements in. In addition to creating the scene, several debugging features are enabled to monitor the performance of the scene. Also, note that the `allowsCameraControl` property on the scene view is set to `true`. This will allow users to move a virtual camera around so they can explore the scene by swiping around in it.

Every SceneKit scene is viewed as if you're looking at it through a camera. You will need to add this camera to the scene yourself, and you must set it up appropriately for your purpose. The fact that SceneKit uses a camera is very convenient because the camera that you are going to set up in a second is replaced by the actual camera of a device when the scene is run with ARKit.

Add the following lines of code to `viewDidLoad()` to create and configure the camera:

```
let cameraNode = SCNNNode()  
cameraNode.camera = SCNCamera()  
cameraNode.position = SCNVector3(x: 0, y: 0, z: 15)  
scene.rootNode.addChildNode(cameraNode)
```

Setting up a basic camera isn't very complicated. All you need is an `SCNNNode` to add the camera to and an `SCNCamera` that will be used to view your scene through. Note that the camera is positioned using an `SCNVector3` object. All nodes in a SceneKit scene use this object to express their positions in 3D space.

In addition to using a simulated camera, SceneKit also simulates real lighting conditions. When you run your scene with ARKit, the lighting conditions will be automatically managed by ARKit, making your objects look as if they truly are part of the environment. When you create a plain scene, however, you will need to add the lights yourself. Add the following lines of code to implement some ambient lighting:

```
let ambientLightNode = SCNNNode()  
ambientLightNode.light = SCNLight()  
ambientLightNode.light!.type = .ambient  
ambientLightNode.light!.color = UIColor.orange  
scene.rootNode.addChildNode(ambientLightNode)
```

You can add different types of lights to a SceneKit scene. You can use ambient light as this sample does, but you can also add directional lights that focus on a particular direction, spotlight, or light points that light in all directions.

Now that you have lighting and a camera in place, you can add an object to the scene. You can use several pre-made shapes, also known as geometries, in your scene. Alternatively, you could import an entire 3D model in your scene. If you take a look at the default SceneKit app that Xcode generates if you create a new project with the **Game** template, you can see that it imports a 3D model of an airplane.

In the AR gallery you will build later, the artwork is augmented with digital information signs that are attached to the piece of art they belong to. To practice building such a sign, you will add a rectangular shape, or plane, to your SceneKit scene and place some text on top of it.

Add the following code to create a simple white plane, a node that renders the plane, and add it to the scene:

```
let plane = SCNPlane(width: 15, height: 10)  
plane.firstMaterial?.diffuse.contents = UIColor.white
```

```
plane.firstMaterial?.isDoubleSided = true
plane.cornerRadius = 0.3

let planeNode = SCNNNode(geometry: plane)
planeNode.position = SCNVector3(x: 0, y: 0, z: -15)
scene.rootNode.addChildNode(planeNode)
```

If you were to build and run your app now, you would see a white square that is positioned in front of the camera. By swiping on the scene, you can make the camera move around the plane to view it from all possible sides. Note that the plane appears to be quite large even though it was only set to be 15 wide and 10 height. You might have guessed that these numbers represent points on the screen, just like in other apps. In SceneKit, there is no concept of points. All values for size and distance must be specified in meters. This means that everything you do is done relative to other objects or their real-world sizes. Using real sizes is essential when you take your SceneKit knowledge to ARKit.

To add some text to the plane you just created, use the following code:

```
let text = SCNText(string: "Hello, world!", extrusionDepth: 0)
text.font = UIFont.systemFont(ofSize: 2.3)
text.isWrapped = true
text.containerFrame = CGRect(x: -6.5, y: -4, width: 13, height: 8)
text.firstMaterial?.diffuse.contents = UIColor.red

let textNode = SCNNNode(geometry: text)
planeNode.addChildNode(textNode)
```

The preceding code creates a text geometry. Since all values in SceneKit are in meters, the text size will be a lot smaller than you would probably expect. To make sure the text is positioned properly in the plane, text wrapping is enabled, and a containerFrame is used to specify the bounds for the text. Since the origin for the text field will be in the center of the plane it is displayed on, the x and y positions are offset negatively from the center to make sure the text appears in the correct place. You can try to play with this frame to see what happens. After configuring the text, it is added to a node, and the node is added to the plane node.

If you run your app now, you can see the *Hello, World!* text rendered on the white plane you created before. This sample is an excellent taste of what you're going to create next. Let's dive straight into building your AR gallery!

Implementing an Augmented Reality gallery

Creating an excellent AR experience has been made a lot simpler with the great features that exist in ARKit. However, there still are several things to keep in mind if you want to build an AR experience that users will love.

Certain conditions, such as lighting, the environment, and even what the user is doing, can have an impact on the AR experience. In this section, you will implement an AR gallery, and you will discover firsthand how ARKit is both amazingly awesome and sometimes a little bit fragile.

First, you'll set up a session in ARKit so you can implement image tracking to discover certain predefined images in the world, and you'll show some text above the found picture. Then, you'll implement another feature that allows users to place art from a gallery in the app in their own room.

If you want to follow along with the steps to implement the ARKit gallery, make sure to grab the **ARGallery** start project from the book's code bundle. Before you move on to implementing the AR gallery, explore the starter project for a little bit. The user interface that is prepared contains an instance of `ARSCNView`; this is the view that will be used to render the AR experience. A collection view has been added in preparation of the user adding their own images to the gallery, and a view for error messages was added to inform the user about certain things that might be wrong.

You'll find that the project is quite basic so far. All the existing code does is set up the collection view, and some code was added to handle errors during the AR session. Let's implement image tracking, shall we?

Adding image tracking

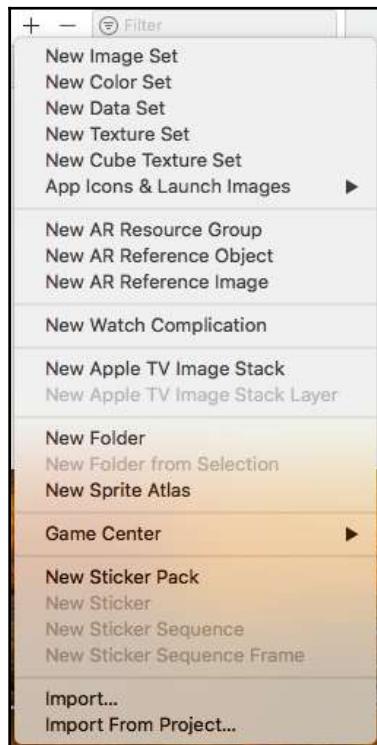
When you add image tracking to your ARKit app, it will continuously scan the environment for images that match the ones you added to your app. This feature is great if you want users to look for specific images in their environment so you can provide more information about them or as part of a scavenger hunt. But more elaborate implementations might exist as part of a textbook or magazine where scanning a particular page would cause the whole page to come alive as part of a unique experience.

Before you can implement the image-tracking experience, you must prepare some images for your users to find in the app. Once the content is ready, you're ready to build the AR experience itself.

Preparing images for tracking

Adding images to your app that are eligible for image tracking is relatively straightforward. The most important part is that you pay close attention to the images you add to your app. It's up to you to make sure that the images you add are high-quality and well-saturated. ARKit will scan for special features in an image to try to match it, so it's important that your image has enough details, contrast, and colors. An image of a smooth gradient might look like a recognizable image to you, but it could be tough for ARKit to detect.

To add images to your project, go to the `Assets.xcassets` folder, click the `+` icon in the bottom-left corner, and select **New AR Resource Group**, as shown in the following screenshot:



After adding a new resource group, you can drag images into the folder that was created. Each resource group will be loaded and monitored by ARKit all at once, so make sure you don't add too many images to a single resource group because that could negatively impact the performance of your app. Apple recommends you add up to about 25 images to a single resource group.

After you add an image to a resource group, Xcode will analyze the images and warn you if it thinks something is wrong with your image. Usually, Xcode will inform you as soon as you add a new image because ARKit requires the physical size of the image you want to detect to be known. So if you're going to detect a specific painting or a page in a magazine, you must add the dimensions for these resources in centimeters as they exist in the real world.

The start project from the code bundle comes with a couple of prepared images that you can explore to see some examples of the kinds of images that you could use in your own apps.



If you want to have some content of your own, take photos of artwork or pictures that you have around the house or office. You can use the Measure app in iOS 12 to measure the physical dimensions of the pictures and add them to your AR gallery project. Make sure that your pictures are well-saturated and free from any glare or reflections.

Once you have found and added some excellent content to use in your AR gallery, it's time to build the experience itself.

Building the image-tracking experience

To implement image tracking, you will set up an `ARSession` that uses `ARWorldTrackingConfiguration` to detect images and track a user's movement through the environment. When one of the images you have prepared is discovered in the scene, an `SCNPlane` will be added above the picture with a short description of the picture itself.

Because ARKit uses the camera, your app must explicitly provide a reason for accessing the camera, so the user understands why your app needs permission to use their camera. Add the `NSCameraUsageDescription` key to the `Info.plist` file and add a short text about why the gallery needs access to the camera.

If you open `ViewController.swift`, you will find a property called `artDescriptions`. Make sure to update this dictionary with the names of the images you added to the resource group, and add a short description for each image.

Next, update `viewDidLoad()` so `ViewController` is set as the delegate for both `ARSCNView` and `ARSession`. Add the following lines of code to do this:

```
arKitScene.delegate = self  
arKitScene.session.delegate = self
```

The scene delegate and session delegate are very similar. The session delegate provides very fine-grained control of the content that is displayed in the scene, and you'll usually use this protocol extensively if you build your own rendering. Since the AR gallery is rendered using SceneKit, the only reason to adopt `ARSessionDelegate` is to respond to changes in the session's tracking state.

All of the interesting methods that you should adopt are part of `ARSCNViewDelegate`. This delegate is used to respond to specific events. For instance, when new features are discovered in the scene or when new content was added.

Currently, your AR gallery doesn't do much. You must configure the `ARSession` that is part of the scene to begin using ARKit. The best moment to set this all up is right before the view controller becomes visible. Therefore, you should do all of the remaining setup in `viewWillAppear(_:)`. Add the following implementation for this method to `ViewController`:

```
override func viewWillAppear(_ animated: Bool) {  
    super.viewWillAppear(animated)  
  
    // 1  
    let imageSet = ARReferenceImage.referenceImages(inGroupNamed: "Art",  
bundle: Bundle.main)!  
  
    // 2  
    let configuration = ARWorldTrackingConfiguration()  
    configuration.planeDetection = [.vertical, .horizontal]  
    configuration.detectionImages = imageSet  
  
    // 3  
    arKitScene.session.run(configuration, options: [])  
}
```

The first step in this method is to read the reference image from the app bundle. These are the images you added to `Assets.xcassets`. Next, `ARWorldTrackingConfiguration` is created, and it's configured to track both horizontal and vertical planes, as well as the reference images. Lastly, the configuration is passed to the session's `run (_:options:)` method. If you run your app now, you should already be prompted for camera usage, and you should see the error-handling working. Try covering the camera with your hand, that should make an error message appear.

Keeping an AR session alive if a view isn't visible anymore is quite wasteful, so it's a good idea to pause the session if the app is closed or if the view controller that contains the AR scene becomes invisible. Add the following method to `ViewController` to achieve this:

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    arKitScene.session.pause()
}
```

In the current setup, the AR session detects your images, but it does nothing to visualize this. When one of the images you added is identified, `ARSCNViewDelegate` is notified of this. To be specific, the `renderer (_:didAdd:for:)` method is called on the scene delegate when a new `SCNNode` is added to the view. For instance, when the AR session discovers a flat surface, it adds a node for `ARPlaneAnchor`, or when it detects one of the images you're tracking, a node for `ARImageAnchor` is added. Since this method can be called with different reasons, it's essential that you add logic to differentiate between the various reasons that could cause a new `SCNNode` to be added to the scene.

Because the AR gallery will implement several other features that could trigger the addition of a new node, you should separate the different actions you want to take for each different type of anchor into specialized methods. Add the following method to `ARSCNViewDelegate` to add the information plane next to a detected image:

```
func placeImageInfo(withNode node: SCNNode, for anchor: ARImageAnchor) {
    let referenceImage = anchor.referenceImage

    // 1
    let infoPlane = SCNPlane(width: 15, height: 10)
    infoPlane.firstMaterial?.diffuse.contents = UIColor.white
    infoPlane.firstMaterial?.transparency = 0.5
    infoPlane.cornerRadius = 0.5
```

```
// 2
let infoNode = SCNNode(geometry: infoPlane)
infoNode.localTranslate(by: SCNVector3(0, 10, -
referenceImage.physicalSize.height / 2 + 0.5))
infoNode.eulerAngles.x = -pi / 4

// 3
let textGeometry = SCNText(string:
artDescriptions[referenceImage.name ?? "flowers"], extrusionDepth:
0.2)
textGeometry.firstMaterial?.diffuse.contents = UIColor.red
textGeometry.font = UIFont.systemFont(ofSize: 1.3)
textGeometry.isWrapped = true
textGeometry.containerFrame = CGRect(x: -6.5, y: -4, width: 13,
height: 8)

let treeNode = SCNNode(geometry: textGeometry)

// 4
node.addChildNode(infoNode)
infoNode.addChildNode(treeNode)
}
```

The preceding code should look somewhat familiar to you. First, an instance of `SCNPlane` is created. Then, this plane is added to `SCNNode`. This node is translated slightly to position it above the detected image. This translation uses `SCNVector3` so it can be translated into three dimensions. The node is also rotated a little bit to create a nice-looking effect.

Next, add the following implementation for `renderer(_:didAdd:for:)`:

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for
anchor: ARAnchor) {
    if let imageAnchor = anchor as? ARImageAnchor {
        placeImageInfo(withNode: node, for: imageAnchor)
    }
}
```

This method checks whether the anchor that was discovered is an image anchor; if it is, `placeImageInfo(withNode:for:)` is called to display the information sign.

Go ahead and run your app now! When you find one of the images that you added to your resource group, an information box should appear on top of it as shown in the following screenshot:



Pretty awesome, right? Let's take it one step further and allow users to position some of the pictures from the collection view wherever they want in the scene.

Placing your own content in 3D space

To spice up the AR gallery a little bit, it would be great to be able to add some new artwork to the environment. Using ARKit, doing this becomes relatively simple. There are a couple of gotchas to take into account when implementing a feature such as this, but overall Apple did a great job making ARKit an accessible platform to work with for developers.

When a user taps on one of the images in the collection view at the bottom of the screen, the image they tapped should be added to the environment. If possible, the image should be attached to one of the walls surrounding the user. If this isn't possible, the image should still be added, except it will float in the middle of the space.

To build this feature, you should implement

`collectionView(_:didSelectItemAt:)` since this method is called when a user taps on one of the items in a collection view. When this method is called, the code should take the current position of the user in the environment and then insert a new `ARAnchor` that corresponds to the location where the new item should be added.

Also, to detect nearby vertical planes, such as walls, some hit testing should be done to see whether a vertical plane exists in front of the user. Add the following implementation of `collectionView(_:didSelectItemAt:)` to implement this logic:

```
func collectionView(_ collectionView: UICollectionView,
didSelectItemAt indexPath: IndexPath) {
    // 1
    guard let camera = arKitScene.session.currentFrame?.camera
    else { return }

    // 2
    let hitTestResult = arKitScene.hitTest(CGPoint(x: 0.5, y: 0.5),
types: [.existingPlane])
    let firstVerticalPlane = hitTestResult.first(where: { result in
        guard let planeAnchor = result.anchor as? ARPlaneAnchor
        else { return false }

        return planeAnchor.alignment == .vertical
    })

    // 3
    var translation = matrix_identity_float4x4
    translation.columns.3.z = -Float(firstVerticalPlane?.distance ?? -1)
    let cameraTransform = camera.transform
    let rotation = matrix_float4x4(cameraAdjustmentMatrix)
    let transform = matrix_multiply(cameraTransform,
matrix_multiply(translation, rotation))

    // 4
    let anchor = ARAnchor(transform: transform)
    imageNodes[anchor.identifier] = UIImage(named:
images[indexPath.row])!
```

```
    arKitScene.session.add(anchor: anchor)
}
```

Even though there are only four steps in this snippet, a lot is going on. First, the camera is grabbed from the current frame in the AR session so it can be used later to determine the user's location in the scene.

Next, a hit test is performed to see whether any planes were already detected in the scene. Since this hit test will return both vertical and horizontal planes, the results are filtered to find the very first vertical plane that was found in the hit test.

Since the location of every `ARAnchor` is represented as a transformation from the world origin, the third step is to determine the transformation that should be applied to position the new artwork in the correct place. The world origin is the place where the AR session first became active.

After creating a default translation, the `z` value for the translation is adjusted, so the object is added either in front of the user or against the nearest vertical plane. Next, the current position of the user is retrieved through the camera. The rotation for the camera will have to be adjusted in the next steps because the camera does not follow the device's orientation. This means that the camera will always assume that the `x`-axis runs across the length of the device, starting at the top and moving downward towards the home indicator area. A computed property to determine how the rotation should be adjusted is already added to the AR gallery starter project.

After setting up the correct transformation properties for the anchor, an instance of `ARAnchor` is created. The unique identifier and image that the user tapped are then stored in the `imageNodes` dictionary so the image can be added to the scene after the new anchor is registered on the scene.

To add the image to the scene, you should implement a helper method that will be called from `rendered(_:_didAdd:for:)`, similar to the helper method you added to show the information card for the image-tracking feature. Add the following code to `ViewController` to implement this helper:

```
func placeCustomImage(_ image: UIImage, withNode node: SCNNNode) {
    let plane = SCNPlane(width: image.size.width / 1000, height:
        image.size.height / 1000)
    plane.firstMaterial?.diffuse.contents = image

    node.addChildNode(SCNNode(geometry: plane))
}
```

To make it easier to see whether an appropriate vertical plane exists, you can implement a helper method that visualizes the planes that the AR session discovers. Add the following code to the `ViewController` class to implement this helper:

```
func vizualise(_ node: SCNNNode, for planeAnchor: ARPlaneAnchor) {  
    let infoPlane = SCNPlane(width: CGFloat(planeAnchor.extent.x),  
    height: CGFloat(planeAnchor.extent.z))  
    infoPlane.firstMaterial?.diffuse.contents = UIColor.orange  
    infoPlane.firstMaterial?.transparency = 0.5  
    infoPlane.cornerRadius = 0.2  
  
    let infoNode = SCNNNode(geometry: infoPlane)  
    infoNode.eulerAngles.x = -.pi / 2  
  
    node.addChildNode(infoNode)  
}
```

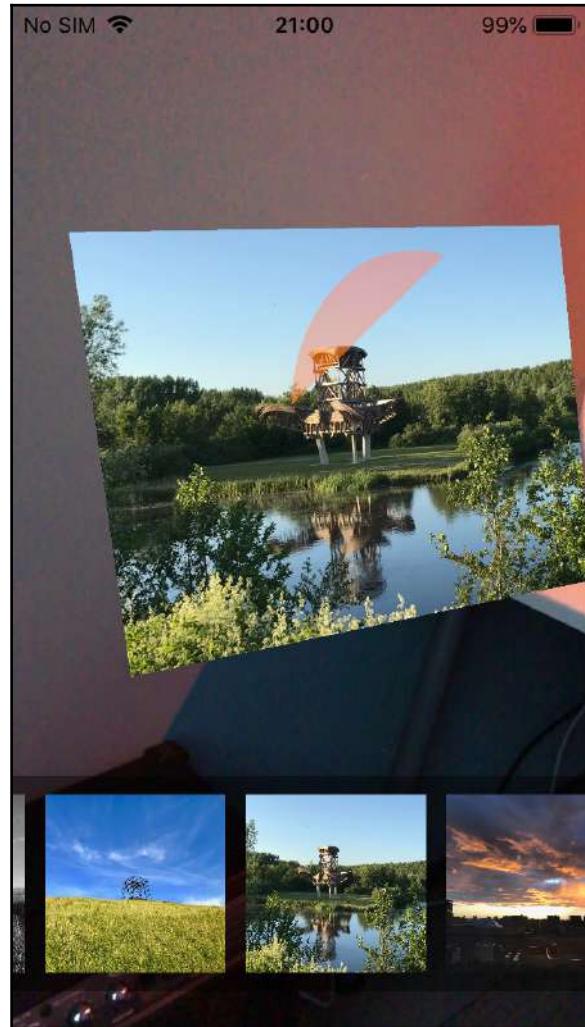
The previous method takes a node and anchor to create a new `SCNPlane`, which is added to the exact position where the new plane anchor was discovered.

The final step to implementing this feature is to call the helper methods when needed. Update the implementation for `renderer(_:didAdd:for:)` as follows:

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNNode, for  
anchor: ARAnchor) {  
    if let imageAnchor = anchor as? ARImageAnchor {  
        placeImageInfo(withNode: node, for: imageAnchor)  
    } else if let customImage = imageNodes[anchor.identifier] {  
        placeCustomImage(customImage, withNode: node)  
    } else if let planeAnchor = anchor as? ARPlaneAnchor {  
        vizualise(node, for: planeAnchor)  
    }  
}
```

If you run your app now, you should see orange squares appear in areas where ARKit detected a flat surface. Note that ARKit needs textures and visual markers to work well. If you try to detect a solid white wall, it's unlikely that ARKit will properly recognize the wall due to a lack of textures. However, a brick wall or a wall that has wallpaper with some graphics on it should work well for this purpose.

The following image shows an example where an image is attached to a wall, together with the plane indicator:



This wraps up the implementation of your own personal AR gallery. There still is much to learn about the things you can do with AR, so make sure to keep on experimenting and learning so you can create amazing experiences for your users.

Summary

In this chapter, you learned a lot. You gained some insight into what AR is, the basic workings of AR, and what you can do with it. Then you learned about the components that make up an excellent AR experience, and you implemented your first small AR experience by adopting Quicklook in an app to preview AR content in a real AR session.

Then you explored different ways to render content in an AR scene. You took a quick look at SpriteKit and SceneKit, and learned that SpriteKit is Apple's 2D game-development framework. You also learned that SceneKit is Apple's 3D game framework, which makes it extremely suited for usage in an AR app.

Then you implemented an AR gallery that uses image tracking, plane detection, and allows users to add their own contents to their gallery. In the process of doing this, you saw that it's not always easy to get ARKit to work well. Bad lighting and other factors can make AR experiences less than ideal.

In the next chapter, you will learn about location services and how you can use them to create better experiences for your users.

Questions

1. What is SceneKit typically used for?
 - a) 2D games.
 - b) Only for AR.
 - c) 3D games.
2. What is a view in a SpriteKit game called?
 - a) A view.
 - b) A sprite.
 - c) A node.
3. What unit of measurement do SceneKit and ARKit use?
 - a) Points.
 - b) Inches.
 - c) Centimeters.

4. Which tracking mode tracks both device movement and rotation?
 - a) Image tracking.
 - b) Orientation tracking.
 - c) World tracking.
5. What should you keep in mind when you prepare images for image tracking in ARKit?
 - a) The images should be colorful.
 - b) The images should be big.
 - c) The images should have sufficient features.
6. What does ARKit need to accurately track objects?
 - a) A low-light environment.
 - b) Smooth surfaces.
 - c) Surfaces with lots of textures.

Further reading

- Swift 3 Game Development – Stephen Haney: <https://www.packtpub.com/application-development/swift-3-game-development-second-edition>
- 3D iOS Games by Tutorials – Chris Language: <https://store.raywenderlich.com/products/3d-apple-games-by-tutorials>
- Apple's MultipeerConnectivity framework: <https://developer.apple.com/documentation/multipeerconnectivity>

17

Improving Apps With Location Services

All iOS devices come with a huge variety of chips and sensors that can be used to enhance the user experience. Augmented reality applications make heavy use of sensors such as the gyroscope, accelerometer, and camera. These sensors are great if you want to grab a picture or want to know how a device is moving. Other apps require different data, like, for instance, the user's GPS location at a given time. In this chapter, you will learn how to use the `CoreLocation` framework to do just that.

Core Location is a framework that allows developers to gain access to a user's current location, but it also allows developers to track whether a user has entered or exited a specific area, or even to monitor a user's location over time. A proper implementation of Core Location can be the core of many great features in your app, but a lousy implementation could drain a user's battery in no time.

In this chapter, you will learn about the following location-related topics:

- Requesting a user's location
- Subscribing to location changes
- Setting up geofences

If you want to follow along with the code samples and implementations that are provided throughout this chapter, make sure to grab the **LocationServices** start project from this book's code bundle. By the end of the chapter, you should be able to make educated decisions about how and when you implement Core Location in your apps.

Requesting a user's location

As you can imagine, giving an application access to your exact location is quite a big deal. In the wrong hands, this data could allow people with malicious intentions to know exactly where you are at any given time, and abuse this knowledge in many different ways. For this reason, it's essential that you only request a user's location if you absolutely have to. Simply using it for a small feature, or to make sure a user is in some arbitrary location before they sign up for a service, might not always be a good enough reason to ask for a user's location.

Asking for permission to access location data

When you do need access to a user's location, you have to ask permission first. Similar to how you have to add a reason for needing the camera or a user's contacts to the `Info.plist` file, you must also provide a reason for requesting location data. In the case of location data, there are two keys you can add to the `Info.plist`:

- Privacy-Location When In Use Usage Description
(`NSLocationWhenInUseUsageDescription`)
- Privacy-Location Always And When In Use Usage Description
(`NSLocationAlwaysAndWhenInUseUsageDescription`)

When your app asks for permission to use a user's location data, they have the option to only allow your app access to their location when the app is in use, or they can choose to allow your app to access their location all the time, even when the app is in the background. You can also configure the type of access that you want to ask yourself. If you only need the user's location when they are using the app, make sure to configure your permission request properly, so the user isn't asked to provide their location to your app when it's in the background.

After adding the required keys to the `Info.plist` file in the `LocationServices` app, you will need to write some code to ask the user for permission to use their location. Before doing this, let's quickly examine the sample project's structure and content, so you are aware of what information can be found where.

First, open the `Main.storyboard` file in the project. You will find a tab bar controller with two view controllers in it. Throughout this chapter, you will implement the features to populate these view controllers with appropriate data. Next, look at the `AppDelegate` implementation. The implementation here follows the Dependency Injection pattern that you have seen used in earlier projects, so there shouldn't be any surprises for you here. Now, go ahead and examine the view controllers. The `GeofenceViewController` is the one you will work on first, to make the user's current location appear on the screen.

You will notice that a lot of code has already been implemented in this view controller. Examine the existing code for a bit, and you'll find that all the code makes calls to empty methods in `LocationHelper.swift`. Most of your focus in this chapter will be on implementing the Core Location code required to work with a user's location data, so the UI work has already been set up. As you add code to `LocationHelper`, you'll find that the user interface for `LocationServices` comes to life bit by bit.

Now that you have a better understanding of how the `LocationServices` app was set up, let's see what steps are involved in asking the user for permission to use their location. Since this app will eventually track location changes in the background, you should ask the user access to their location even when the app is in the background. To do this, add the following `viewDidAppear(_:)` code to `GeofenceViewController`:

```
locationHelper.askPermission { [weak self] status in
    if status == .authorizedAlways {
        self?.showCurrentLocation()
    } else {
        // handle the case where you don't always have access
    }
}
```

This is the first view controller the user will see, so asking the user for their location as soon as this view appears is a good idea. If it's not obvious that you will be prompting the user for their location, it's often a good idea to inform the user about why you are going to ask them for location permissions before actually showing the location access dialog. To actually make the permission dialog appear, you will need to add some code to `LocationHelper.swift`.

All location service-related requests are performed through an instance of `CLLocationManager`. The location manager is responsible for obtaining the user's GPS location, asking for permission to access the user's location, and more. When the location manager receives updates about the user's location, authorization status, or other events, it will notify its delegate. A location manager delegate should conform to the `CLLocationManagerDelegate` protocol. Note that the `LocationHelper` already conforms to `CLLocationManagerDelegate`, and that an instance of `CLLocationManager` is already created on this object. All that's left to do is assign the helper as the delegate for the location manager. Add the following line at the end of the `init()` method in `LocationHelper` to set it as the location manager delegate:

```
locationManager.delegate = self
```

Next, add the following implementation for the `askPermission(_:)` method:

```
func askPermission(_ completion: @escaping (CLAuthorizationStatus) -> Void) {
    let authorizationStatus = CLLocationManager.authorizationStatus()
    if authorizationStatus != .notDetermined {
        completion(authorizationStatus)
    } else {
        askPermissionCallback = completion
        locationManager.requestAlwaysAuthorization()
    }
}
```

This implementation checks whether a current authorization status exists. If it does, the completion callback is called with the current status. If the current status has not been determined yet, the location manager is asked to request authorization to access the user location using the `requestAlwaysAuthorization()` method. This will prompt the user for their location permissions. The reason you need to have permanent access to a user's location in this app is to ensure you can implement geofencing later in this chapter. Add the following method to the `CLLocationManagerDelegate` to retrieve the user's response to the authorization prompt:

```
func locationManager(_ manager: CLLocationManager,
didChangeAuthorization status: CLAuthorizationStatus) {
    askPermissionCallback?(status)
    askPermissionCallback = nil
}
```

The preceding code immediately passes the user's response to the stored completion callback that was passed to `askPermission(_:)`. After calling the callback, it is set to `nil` to avoid accidentally calling it again. At this point, you have done all the work required to request access to a user's location. Let's see how you can retrieve a user's current location next.

Obtaining a user's location

Once your app has access to location data, you can use the location manager to begin observing a user's location, the direction in which a user is heading, and more. For now, you will focus on obtaining the user's current location. The `GeofenceViewController` already contains a method, called `showCurrentLocation()`, that is responsible for asking the location helper for a current location. If you examine this method closely, you'll find that it also asks the location helper for a location name by calling `getLocationName(for:_:)` and passing the obtained location to this method. The `showCurrentLocation()` method also uses the obtained location to focus a map view on the user's location by calling `setRegion(_:animated:)` on the map view.

Since the view controller is already fully prepared to handle location updates, all you need to do is add the proper implementations for `getLatestLocation(_:)` and `getLocationName(for:_:)`. Begin by adding the following implementation for `getLatestLocation(_:)`:

```
func getLatestLocation(_ completion: @escaping (CLLocation) -> Void) {
    if let location = trackedLocations.last {
        completion(location)
    } else if CLLocationManager.locationServicesEnabled() {
        latestLocationObtainedCallback = completion
        locationManager.startUpdatingLocation()
    }
}
```

The preceding method first checks whether a location has already been obtained. If it has, then the latest obtained location is returned. If there is no existing location, the code checks whether location services are enabled. It's always good practice to check whether the location service you are about to use is actually available. If location services are available, the completion callback is stored in the helper, and the location manager is told to start monitoring the user's location by calling `startUpdatingLocation()`.

Calling `startUpdateLocation()` will make the location observer continuously monitor the user's GPS location, and will send any relevant updates to its delegate by calling `locationManager(_:didUpdateLocations:)`. This method will always receive one or more new locations that the manager has obtained, where the latest location will be the last item in the list of obtained locations. Add the following implementation for this method to the `CLLocationManagerDelegate` extension of `LocationHelper`:

```
func locationManager(_ manager: CLLocationManager, didUpdateLocations
locations: [CLLocation]) {
    latestLocationObtainedCallback?(locations.last!)
    latestLocationObtainedCallback = nil
    locationManager.stopUpdatingLocation()

    trackedLocations += locations
}
```

The implementation for `locationManager(_:didUpdateLocations:)` is fairly straightforward: the latest location is passed to the callback, and the callback is removed to prevent subsequent location updates from triggering the callback unexpectedly. Also, the location manager is told to stop monitoring the user's location by calling `stopUpdatingLocation()`. Lastly, the obtained locations are stored for later use.



It's always good practice to make the location manager stop monitoring location updates if you won't be needing updates any time soon. Monitoring location updates has a pretty significant impact on battery life, so you shouldn't spend more time tracking a user's location than needed.

Now that you can retrieve the user's location, the last step is to also retrieve the location name, by implementing `getLocationName(for:completion:)` in the location helper. Add the following implementation for this method to the location helper:

```
func getLocationName(for location: CLLocation, _ completion: @escaping
(String) -> Void) {
    let geocoder = CLGeocoder()
    geocoder.reverseGeocodeLocation(location) { placemarks, error in
        guard error == nil else {
            completion("An error occurred: \(error?.localizedDescription ??
"Unknown error")")
            return
        }

        completion(placemarks?.first?.name ?? "Unknown location")
```

```
    }  
}
```

The preceding code uses a `CLGeocoder` to find a placemark that corresponds with the user's current location. Note that this feature uses an internet connection, so the name lookup will only work if the user has an internet connection. Regular GPS-related features do not require internet access, so your app can monitor and track a user's location even if they don't have an active internet connection.

Try running your app now—you should be able to see the user's current location on the map, and the location name, latitude, and longitude should be displayed on the screen as well. Now that you know how to obtain a user's location, let's see how you can efficiently subscribe your app to follow changes in a user's location.

Subscribing to location changes

One way of subscribing to changes in a user's location has already been covered in the previous section of this chapter. When you call `startUpdatingLocation()` on a location manager, it will automatically subscribe to the user's location. This method of tracking a user's location is excellent if you need very detailed reporting on a user's location, but usually, you don't need this level of detail. More importantly, using this kind of location tracking for an extended period will drain the user's battery.

Luckily, there are better ways to listen to location changes. One way is to subscribe to locations that the user visits by calling `startMonitoringVisits()`. This method is used if you aren't interested in the user's detailed movement but only want to know whether the user spent an extended period in a particular area. This type of tracking of a user's location is perfect if you need a low-power way to track very rough location changes. This kind of tracking even works well if your app is running in the background, because your app will automatically be woken up or launched if a visit event occurs.

If your app is relaunched due to a location-related event, then `UIApplication.LaunchOptionsKey.location` will be present in the application's launch options dictionary. When it is, you are expected to create an instance of a location manager and assign it a delegate to receive the relevant location update.

If the visits monitoring is a bit too inaccurate for your purposes but you don't need continuous location tracking, you can use significant location change tracking. This type of tracking triggers when a user has moved a significant distance over time, providing your app with updates only when the user is truly moving. This is a lot more power efficient than making your app track the user's location even when their current location hasn't changed. Just like visits tracking, significant location changes will wake up your app or even relaunch it when they occur. When an event is delivered to the app like this, you should re-enable the significant location changes monitoring. Let's implement significant location changes in the LocationServices sample app to see exactly how they work .

If you look at `SignificantChangesViewController`, you'll note that the view controller is fully set up to begin monitoring significant location changes. The `monitorSignificantChanges (_:)` method, defined on the location helper, takes a callback that's called every time a significant location change occurs. Every time new location data is retrieved, the table view is reloaded to display the latest available data. Since significant location updates can wake the app up with a special key in the app's launch options, let's update `AppDelegate` so it can handle this scenario. Add the following code `application (_: didFinishLaunchingWithOptions:)` right before the return statement:

```
if launchOptions?[UIApplication.LaunchOptionsKey.location] != nil {
    locationHelper.monitorSignificantChanges { _ in
        // continue monitoring
    }
}
```

Since `AppDelegate` already has a reference to the location helper, all it needs to do is re-enable significant location changes monitoring. This small change to `AppDelegate` is quite powerful because it allows your app to respond to changes in the user's location even when the app is not running. Let's implement the appropriate code in the location helper next.

Add the following implementation for

`monitorSignificantLocationChanges (_:)` to `LocationHelper`:

```
func monitorSignificantChanges(_ locationHandler: @escaping
(CLLocation) -> Void) {
    guard
    CLLocationManager.significantLocationChangeMonitoringAvailable()
    else { return }

    significantChangeReceivedCallback = locationHandler
    locationManager.startMonitoringSignificantLocationChanges()
```

```
    isTrackingSignificantLocationChanges = true  
}
```

This method is very similar to the location helper methods you have seen before. When a significant location change is detected, the location manager calls `locationManager(_:didUpdateLocations:)` on its delegate. Since this method is already implemented, you should update the implementation as follows:

```
func locationManager(_ manager: CLLocationManager, didUpdateLocations  
locations: [CLLocation]) {  
    latestLocationObtainedCallback?(locations.last!)  
    latestLocationObtainedCallback = nil  
  
    if isTrackingSignificantLocationChanges == false {  
        locationManager.stopUpdatingLocation()  
    }  
  
    significantChangeReceivedCallback?(locations.last!)  
  
    trackedLocations += locations  
}
```

Note that the location manager is only told to stop updating the user's location when significant location tracking is not active. When you call `stopUpdatingLocation()`, the location manager will cease to deliver any location updates to this delegate method. Also, note that `significantChangeReceivedCallback` is not removed after calling it. The reason for this is that the caller of `monitorSignificantChanges(_:)` is interested in continuous location updates, so any time this method is called, the `SignificantChangesViewController` view controller that initiated significant location tracking should always be called.

One last thing you need to do, so your app receives significant location changes while it's not in the foreground, is to set the `allowsBackgroundLocationUpdates` property to `true`. Add the following line of code to the location helper's `init()`:

```
locationManager.allowsBackgroundLocationUpdates = true
```

In addition to subscribing to significant location changes or visits, you can also respond to the user entering or leaving a certain area with geofences.

Setting up geofences

Sometimes, your app doesn't really need to know the details of the user's whereabouts. Sometimes, you're only interested in tracking whether the user has exited or left a certain area, in order to show certain content in your app or to unlock some kind of special feature. Core Location has great support for monitoring geofences. A geofence is a certain area that is defined using a certain GPS coordinate and a circular radius around this point. In Core Location, geofences are set up using `CLRegion` subclasses. Core Location provides two different region types that you can use:

- `CLCircularRegion`
- `CLBeaconRegion`

A `CLCircularRegion` type is used to set up a geofence, as described before. A `CLBeaconRegion` type is used with physical BLE iBeacons, and essentially provides geofencing in a very small radius, for instance, just a couple of meters. In this section you will learn how to set up a `CLCircularRegion` type that is set up around a user's first detected location. Setting up geofencing, or region monitoring, with both types of regions is very similar so all principles for monitoring a circular region also applies to beacon regions.

If you look at the `GeofenceViewController`, you'll notice that it has a button labelled **Set Geofence**. The `@IBAction` for this button does quite a lot of the work already, but one key element is missing—it doesn't inform the location manager about the region that should be monitored. Add the following code to the end of `setGeofence()` in `GeofenceViewController`:

```
let region = CLCircularRegion(center: location.coordinate, radius: 30,  
identifier: "current-location-geofence")  
locationHelper.setGeofence(at: region, exitHandler, enterHandler)
```

The preceding code uses the location that was obtained from the user before, and uses it to create a circular region with a radius of 30 meters. The identifier that is passed to the region should be an identifier that uniquely defines the region. If you reuse an identifier, Core Location will stop monitoring the old region with that identifier, and will monitor the new region instead. For the `LocationServices` app, this is perfect, but if you want your app to observe multiple regions, you must make sure every region has its own unique identifier.

Next, add the following implementation for `setGeofence(at:_:_:)` to the `LocationHelper`:

```
func setGeofence(at region: CLRegion, _ exitHandler: @escaping () -> Void, _ enterHandler: @escaping () -> Void) {
    guard CLLocationManager.isMonitoringAvailable(for: CLCircularRegion.self)
        else { return }

    geofenceExitCallback = exitHandler
    geofenceEnterCallback = enterHandler
    locationManager.startMonitoring(for: region)
}
```

The preceding method is again very similar to the other location helper methods. Let's move right on to implementing the `CLLocationManagerDelegate` methods that the location manager will call:

```
func locationManager(_ manager: CLLocationManager, didEnterRegion
region: CLRegion) {
    geofenceEnterCallback?()
}

func locationManager(_ manager: CLLocationManager, didExitRegion
region: CLRegion) {
    geofenceExitCallback?()
}
```

The preceding two methods are part of the `CLLocationManagerDelegate` protocol, and are called when a user enters or exits a certain area. Since there's no extra work to be done by the helper, the corresponding callbacks are immediately called so the `GeofenceViewController` can update its interface accordingly.

Try opening the app and tapping the **Set Geofence** button. An orange circle should now appear on the map to visualize the geofence you have set up. If you exit or enter the region, the status label should update accordingly, to show whether you have just entered or left the geofence. Note that it might take up to five minutes for iOS to properly register, monitor, and report updates about your geofence. Note that your user should have an active internet connection for region monitoring to work optimally.

Summary

In this chapter, you have learned several techniques to obtain and respond to a user's location. You have implemented a `LocationHelper` class that provided a simple interface for view controllers to use the location manager that is contained in the helper. You learned about the best practices in the area of asking the user for access to their location data, and you have learned that asking for a user's location is a pretty privacy-sensitive question that shouldn't be asked without a good reason.

You learned that there are different ways, each with different levels of detail, that you can use to track a user's location. You saw that you can subscribe to continuous changes, which has a bad impact on battery life. You also learned about subscribing to visits and significant location changes. In addition to learning about tracking a user's location, you also learned about monitoring whether a user has entered or exited a certain area by implementing geofencing. When you implement Core Location in your own apps, always make sure to keep the user's privacy in mind. If you don't really need the location data, then don't request access to it. And if you do, make sure to handle your user's location data with great care.

In the next chapter, you will learn about the CoreML framework and how you can use it to enhance your apps.

Questions

1. What different types of location access exist?
 - a) Always and When in use
 - b) Always and When in background
 - c) When in use and When in background
2. When should you ideally ask for access to location services?
 - a) When your app launches
 - b) As soon as you want access to a user's location
 - c) Once the user understands why you need access to their location
3. How do you obtain a user's current location?
 - a) By calling `getCurrentLocation()` on the location manager
 - b) By calling `startUpdatingLocation()` on the location manager
 - c) By calling `startMonitoring(for:)` on the location manager

4. Which of the following location tracking techniques is the most battery friendly?
 - a) Significant location change tracking
 - b) Continuous location tracking
 - c) Visit tracking
5. What different types of regions can you monitor?
 - a) Geofences and iBeacons
 - b) Geofences and WiFi networks
 - c) WiFi networks and iBeacons
6. Up to how long could it take for region monitoring to start?
 - a) 5 minutes
 - b) 10 minutes
 - c) 2 minutes
7. How can you check whether your app launched due to a significant location change?
 - a) By checking whether a new location is available
 - b) By inspecting the app's launch options
 - c) By implementing a special AppDelegate method

Further reading

- Apple's Core Location documentation can be found at <https://developer.apple.com/documentation/corelocation>

18

Making Smarter Apps with CoreML

Over the past few years, machine learning has gained in popularity. However, it has never been easy to implement in mobile applications—that is, until Apple released the **CoreML** framework as part of iOS 11. CoreML is Apple's solution to all of the problems they have run into themselves while implementing machine learning for iOS. As a result, CoreML should have the fastest, most efficient implementations for working with complex machine learning models, through an interface that is as simple and flexible as possible.

In this chapter, you will learn what machine learning is, how it works, and how you can use trained machine learning models in your apps. You will also learn how you can use Apple's *Vision framework* to analyze images, and you'll see how it integrates with CoreML for powerful image detection. Lastly, you'll learn how to use the new **CreateML** tool to train your own models. You will learn about these topics in the following sections:

- Understanding machine learning and CoreML
- Combining CoreML and computer vision
- Training your own models with CreateML

By the end of this chapter, you will be able to train and use your own CoreML models to make the apps you build more intelligent and compelling.

Understanding machine learning and CoreML

Machine learning and CoreML go hand in hand, but they're not quite the same. Machine learning is all about teaching a machine how it can recognize, analyze, or apply certain things. The result of all this teaching is a trained model that can be used by CoreML to analyze specific inputs and produce an output based on the rules that were established during the training phase.

Before you learn about CoreML, it's good to obtain some knowledge about machine learning to make sure you're familiar with some of the terms that are used, and so you know what machine learning is.

Understanding what machine learning is

A lot of developers will hear about machine learning, deep learning, or neural networks at some point in their career. You may have already heard about these topics. If you have, you know that machine learning is a complex field that requires particular domain knowledge. However, machine learning is becoming more prominent and popular by the day, and it is used to improve many different types of applications.

For instance, machine learning can be used to predict what type of content a particular user might like to see in a music app, based on music that they already have in their library, or to automatically tag faces in photos to connect them to people in the user's contact list. It can even be used to predict costs for specific products or services based on past data. While this might sound like magic, the flow for creating machine learning experiences like these can be split roughly into two phases:

1. Training a model
2. Using inference to obtain a result from the model

Large amounts of high-quality data must be collected to perform the first step. If you're going to train a model that should recognize cats, you will need large amounts of pictures of cats. You must also collect images that do not contain cats. Each image must then be appropriately tagged to indicate whether the image includes a cat or not.

If your dataset only contains images of cats that face towards the camera, the chances are that your model will not be able to recognize cats from a sideways point of view. If your dataset does contain cats from many different sides, but you only collected images for a single breed or with a solid white background, your model might still have a tough time recognizing all cats. Obtaining quality training data is not easy, yet it's essential.

During the training phase of a model, it is imperative that you provide a set of inputs that are of the highest quality possible. The smallest mistake could render your entire dataset worthless. It's in part due to the process of collecting data that training a model is a tedious task. One more reason is that training a model typically takes a lot of time. Certain complex models could take a couple of hours to crunch all the data and train themselves.

A trained model comes in several types. Each type of model is suitable for a different kind of task. For instance, if you are working on a model that can classify certain email messages as spam, your model might be a so-called **support vector machine**. If you're training a model that recognizes cats in pictures, you are likely training a **neural network**.

Each model comes with its own pros and cons, and each model is created and used differently. Understanding all these different models, their implications, and how to train them is extremely hard, and you could likely write a book on each kind of model.

In part, this is why CoreML is so great. CoreML enables you to make use of pre-trained models in your own apps. On top of this, CoreML standardizes the interface that you use in your own code. This means that you can use complex models without even realizing it. Let's learn more about CoreML, shall we?

Understanding CoreML

Due to the complex nature of machine learning and using trained models, Apple has built CoreML to make incorporating a trained model as simple as possible. On top of this, another goal was to ensure that whenever you implement machine learning using CoreML, your implementation is as fast and energy efficient as possible. Since Apple has been enhancing iOS with machine learning for a couple of years now, they have loads of experience of implementing complex models in apps.

If you have ever researched machine learning, you might have come across cloud-based solutions. Typically, you send a bunch of data to such a cloud-based solution, and the result is passed back as a response to your request. CoreML is very different, since the trained model lives on the device, instead of in the cloud. This means that your user's data never has to leave the device, which is very good for your user's privacy. Also, having your trained model on the device means that no internet connection is required to use CoreML, which saves both time and precious data. And since there is no potential bottleneck regarding response latency, CoreML is capable of calculating results in real time.

In the previous section, you learned that there are several types of trained models. Each type of model is used slightly differently, so if you were to implement machine learning in your app manually, you would have to write different wrappers around each of the different models your app uses. CoreML makes sure that you can use each type of model without even being aware of this in your app; they all share the same programming interface. A CoreML model is domain agnostic.

To be domain agnostic, all trained models that you use with CoreML must be in a particular format. Since machine learning already has a vibrant community with several popular formats, Apple has made sure that the most popular models can be easily converted to Apple's own `.mlmodel` format. Let's see how to obtain `.mlmodel` files for you to use in your own apps.

Obtaining CoreML models

There are two ways to obtain a model for you to use in your apps. The simplest way is to find an existing `.mlmodel` file. You can find several ready-to-use `.mlmodel` files on Apple's machine learning website, at <https://developer.apple.com/machine-learning/>. This website contains several of the most popular models. At the time of writing, most of these models are focused on recognizing dominant objects in an image, and chances are that you have different needs for your app.

If you're looking for something that isn't already converted by Apple, you can try to look in several places online for a pre-converted `.mlmodel` file, or you can convert an existing model you have found online. Apple has created converters for several popular machine learning formats, such as *caffe*. The conversion tools for converting an existing model to a `.mlmodel` file are written in Python, and they ship as part of Xcode. If your needs do not fit the converters that Apple provides, you can extend the **toolchain**, since the conversion tools are open source. This means that everybody can add their own converters, or tweak existing converters.

Converting CoreML models using Apple's tools can usually be done with a couple of lines of Python. Writing a good conversion script does typically involve a little bit of domain knowledge in the area of machine learning, because you'll need to make sure that the converted model works just as well as the original model.

Once you have obtained a CoreML model for your app, either by converting one or finding an existing one, you're ready to add it to your project and begin using it. Let's see how to do this next.

Using a CoreML model

Applications can utilize CoreML for many different purposes. One of these purposes is text analysis. You can use a trained model to detect whether a particular piece of text has a positive or negative sentiment. To implement a feature like this, you can use a trained and converted CoreML model.

The code bundle for this chapter includes a project named `TextAnalyzer`. If you open the start version of this project, you'll find a project that has an implementation of a simple layout along with a button that is hooked up to an `@IBAction`, named `analyze()`. The project folder also contains a file called `SentimentPolarity.mlmodel`. This file is a trained CoreML model that analyzes the sentiment associated with a certain text. Drag this file into Xcode to add the CoreML model to your project.

After adding the model to your project, you can click it in the **Project Navigator** to see more information about the model, as illustrated in the following screenshot:

The screenshot shows the Xcode Project Navigator with a selected machine learning model. The model details are as follows:

- Name:** SentimentPolarity
- Type:** Pipeline Classifier
- Size:** 275 KB
- Author:** Vadym Markov
- Description:** Sentiment polarity LinearSVC.
- License:** MIT

In the **Model Class** section, there is a single entry:

- SentimentPolarity** (Swift)

A note indicates it is an "Automatically generated Swift model class".

Under **Model Evaluation Parameters**, the inputs and outputs are listed:

Name	Type	Description
Inputs		
input	Dictionary (String → Double)	Features extracted from the text.
Outputs		
classLabel	String	The most likely polarity (positive or negative), for the given input.
classProbability	Dictionary (String → Double)	The probabilities for each class label, for the given input.

You can see that this model is provided by *Vadym Markov* under the **MIT** license. In the bottom section, you can find out which **inputs** and **outputs** you can expect this model to work with. In this case, the **input** is a dictionary of the `[String: Double]` type. This means that we should feed this model a dictionary of word counts. If you add this model to Xcode, the center section that lists the **Model Class** might notify you that the model isn't part of any targets yet. If this is the case, fix it as you have done previously, by adding this model to your app target in the **Utilities** sidebar on the right side of the window.

Now that your model is implemented, it's time to take it for a spin. First, implement a method that extracts the word count from any given string. You can implement this using the `NLTokenizer` object from the new NaturalLanguage framework. `NLTokenizer` is a text analysis class that is used to split a string into words, sentences, paragraphs, or even whole documents. In this example, the tokenizer is set up to detect individual words. Implement the word count method as follows:

```
func getWordCounts(from string: String) -> [String: Int] {
    let tokenizer = NLTokenizer(unit: .word)
    tokenizer.string = string
```

```
var wordCount = [String: Int]()

tokenizer.enumerateTokens(in: string.startIndex..
```

The previous code iterates over all the words that the tokenizer has recognized, and stores it in a dictionary of the `[String: Double]` type. You might wonder why a `Double` type is used for the word count, rather than an `Int` type, since the word counts won't have to deal with decimals. This is true, but the `SentimentPolarity` model requires its input to be a dictionary of the `[String: Double]` type, so you must prepare the data accordingly.

Now that you have the code to prepare the input data for the `SentimentPolarity` model, let's see how you can use this model to analyze the user's input. Add the following implementation for the `analyze()` method:

```
@IBAction func analyze() {
    let wordCount = getWordCounts(from: textView.text)
    let model = SentimentPolarity()

    guard let prediction = try? model.prediction(input: wordCount)
        else { return }

    let alert = UIAlertController(title: nil, message: "Your text is
rated: \(prediction.classLabel)", preferredStyle: .alert)
    let okayAction = UIAlertAction(title: "Okay", style: .default,
handler: nil)
    alert.addAction(okayAction)
    present(alert, animated: true, completion: nil)
}
```

You might be surprised that this method is so short, but that's how simple CoreML is! First, we retrieve the `wordCount` using the method we implemented earlier. Then, an instance of the CoreML model is created. When you added the `SentimentPolarity` model to the app target, Xcode generated a class interface that abstracted away all complexities involving the model. Because the model is now a simple class, you can obtain a prediction for the sentiment of the text by calling `prediction(input:)` on the model instance.

The `prediction` method returns an object that contains the processed prediction in the `classLabel` property, as well as an overview of all available predictions and how certain the model is about each option in the `classProbability` property. You can use this property if you want to be a bit more transparent to the user about the different options that the model suggested and how certain it was about these options.

In the last section of this chapter, you will learn how you can use **CreateML** to train your own natural language model to analyze texts that use domain-specific language relevant to your own app.

Using CoreML to perform text analysis was quite simple. Now let's see how you can use computer vision together with CoreML to determine the type of object that exists in a particular picture.

Combining CoreML and computer vision

When you're developing an app that works with photos or live camera footage, there are several things you might like to do using computer vision. For instance, it could be desirable to detect faces in an image. Or, maybe you would want to identify certain rectangular areas of photographs, such as traffic signs. You could also be looking for something more sophisticated, like detecting the dominant object in a picture.

To work with computer vision in your apps, Apple has created the **Vision** framework. You can combine Vision and CoreML to perform some pretty sophisticated image recognition. Before you implement a sample app that uses dominant object recognition, let's take a quick look at the Vision framework, so you have an idea of what it's capable of and when you might like to use it.

Understanding the Vision framework

The Vision framework is capable of many different tasks that revolve around computer vision. It is built upon several powerful deep learning techniques to enable state-of-the-art facial recognition, text recognition, barcode detection, and more. When you use Vision for facial recognition, you get much more information than just the location of a face in an image. The framework can recognize several facial landmarks, such as eyes, noses, or mouths. All this is possible due to the extensive use of deep learning behind the scenes at Apple.

For most tasks, using Vision consists of three stages:

1. You create a request that specifies what you want. For instance, a `VNDetectFaceLandmarksRequest` request to detect facial features.
2. You set up a handler that can analyze the images.
3. The resulting observation contains the information you need.

The following code sample illustrates how you might find facial landmarks in an image:

```
let handler = VNImageRequestHandler(cgImage: image, options: [:])
let request = VNDetectFaceLandmarksRequest(completionHandler: {
    request, error in
    guard let results = request.results as? [VNFaceObservation]
        else { return }

    for result in results where result.landmarks != nil {
        let landmarks = result.landmarks!

        if let faceContour = landmarks.faceContour {
            print(faceContour.normalizedPoints)
        }

        if let leftEye = landmarks.leftEye {
            print(leftEye.normalizedPoints)
        }

        // etc
    }
})

try? handler.perform([request])
```

For something as complex as detecting the contour of a face or the exact location of an eye, the code is quite simple. You set up a handler and a request. Next, the handler is asked to perform one or more requests. This means that you can run several requests on a single image.

In addition to enabling computer vision tasks like this, the Vision framework also tightly integrates with CoreML. Let's see just how tight this integration is, by adding an image classifier to the augmented reality gallery app you have been working on!

Implementing an image classifier

The code bundle for this chapter contains an app called **ImageAnalyzer**. This app uses an image picker to allow a user to select an image from their photo library to use it as an input for the image classifier you will implement. Open the project and explore it for a little bit to see what it does and how it works. Use the starter project if you want to follow along with the rest of this section.

To add an image classifier, you need to have a CoreML model that can classify images. On Apple's machine learning website (<https://developer.apple.com/machine-learning/build-run-models/>) there are several models available that can do image classification. An excellent lightweight model you can use is the **MobileNet** model; go ahead and download it from the machine learning page. Once you have downloaded the model, drag the model into Xcode to add it to the **ImageAnalyzer** project. Make sure to add it to your app target so that Xcode can generate the class interface for the model.

After adding the model to Xcode, you can open it to examine the **Model Evaluation Parameters**. The parameters tell you the different types of inputs and outputs the model will expect and provide. In the case of MobileNet, the input should be an image that is 224 points wide and 224 points high, as shown in the following screenshot:



A screenshot of a table titled "Model Evaluation Parameters". The table has three columns: "Name", "Type", and "Description". It is divided into sections for "Inputs" and "Outputs".

Name	Type	Description
Inputs		
image	Image (Color 224 x 224)	Input image to be classified
Outputs		
classLabelProbs	Dictionary (String → Double)	Probability of each category
classLabel	String	Most likely image category

After generating the model, the code to use the model is very similar to the code used to detect facial features with Vision earlier. The most significant difference is that the type of request that is used is a special `VNCoreMLRequest`. This type of request takes the CoreML model you want to use, in addition to a completion handler.

When combining CoreML and Vision, Vision will take care of image scaling and converting the image to a type that is compatible with the CoreML model. You should make sure that the input image has the correct orientation. If your image is rotated in an unexpected orientation, CoreML might not be able to analyze it correctly.

Add the following implementation for `analyzeImage(_:)` to the `ViewController` class in the **ImageAnalyzer** project:

```
func analyzeImage(_ image: UIImage) {
    guard let cgImage = image.cgImage,
        let classifier = try? VNCoreMLModel(for: MobileNet().model)
        else { return }

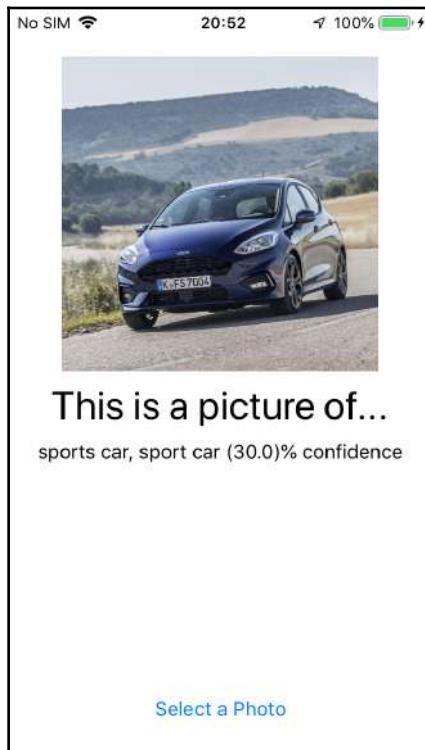
    let request = VNCoreMLRequest(model: classifier, completionHandler:
    { [weak self] request, error in
        guard let classifications = request.results as?
        [VNClassificationObservation],
            let prediction = classifications.first
        else { return }

        DispatchQueue.main.async {
            self?.objectDescription.text = "\(prediction.identifier)
(\(round(prediction.confidence * 100))% confidence"
        }
    })

    let handler = VNImageRequestHandler(cgImage: cgImage, options: [:])
    try? handler.perform([request])
}
```

The previous method takes a `UIImage` and converts it to a `CGImage`. Also, a `VNCoreMLModel` is created, based on the `MobileNet` model. This particular model class wraps the CoreML model, so it works seamlessly with Vision. The request is very similar to the request you have seen before. In the `completionHandler`, the results array and first prediction of the image classifications are extracted and shown to the user. Every prediction made by the classifier will have a label that's stored in the `identifier` and a confidence rating with a value between 0 and 1 stored in the `confidence` property. Note that the value of the description label is set on the main thread to avoid crashes.

You have already implemented two different types of CoreML models that were trained for general purposes. Sometimes, these models won't be specific enough for your purposes. For instance, take a look at the following screenshot, where a machine learning model labels a certain type of car as a sports car with only 30% confidence:



In the next section, you will learn how to train models for purposes that are specific to you and your apps by using CreateML.

Training your own models with CreateML

As part of Xcode 10 and Apple's latest version of macOS, called **Mojave**, they have shipped a new tool that you can use to train your own machine learning models by adding specializations to existing models. This means that you can train your own natural language model that places certain texts in categories that you define. Or, you can train a model that recognizes certain product names or terms in a text that is specific to your application's domain.

If you're building a news app, you might want to train a CoreML model that can automatically categorize the articles in the app. You can then use this model to keep track of the articles your users read, and present articles that are most likely to fit their interests on a dedicated page in your app.

In this segment, you will learn how to train natural language models and how you can train an image recognition model based on the Vision framework. In doing so, you will find that creating a large and optimized training set is crucial when you want to train a machine learning model.

In the code bundle for this chapter, you will find a Playground called `CreateML`. This playground contains all the resources used for training the natural language models.

Training a natural language model

The Natural Language framework has excellent features to analyze text with. Bundled with the power of machine learning models, you can perform some powerful operations on text. Apple has spent a lot of time training several models with vast amounts of data to ensure that the natural language framework can detect names, places, and more.

However, sometimes you might want to add your own analysis tools. To facilitate this, Natural Language works well with CoreML and Apple's new CreateML framework. With CreateML, you can easily and quickly create your own machine learning models that you can use in your apps straight away.

You can use several different types of training for a Natural Language model. In this section, you will learn about two different models:

- A text classifier
- A word tagger

The **text classifier** will classify a particular piece of text with a label. This is similar to the sentiment analysis you have implemented in the **TextAnalyzer** sample app. An example of an entry in your training data would look as follows:

```
{  
    "text": "We took an exclusive ride in a flying car",  
    "label": "Tech"  
}
```

This is a sample of a news article headline that belongs in a category labeled *Tech*. When you feed a large number of samples like this to your model, you could end up with a classifier that can apply labels to news articles based on their headlines. Of course, this assumes that the headlines are specific enough and contain enough information to train the classifier properly. In reality, you will find that short sentences like these will not make the best models. The sample Playground contains a JSON file with training data that attempts to separate news articles into two categories; politics and tech. Let's see how the model can be trained so you can then see for yourself how accurate the model is.

The following code trains and stores the custom CoreML model:

```
import CreateML  
import Foundation  
  
let trainingData = try! MLDataTable(contentsOf:  
Bundle.main.url(forResource: "texts", withExtension: "json")!)  
let model = try! MLTextClassifier(trainingData: trainingData,  
textColumn: "text", labelColumn: "label")  
try! model.write(to: URL(fileURLWithPath:  
"/path/to/folder/TextClassifier.mlmodel"))
```

Training the entire model requires only a couple of lines of code. All you need to do is obtain your training data, create the classifier, and save it somewhere on your machine. You can even do some quick testing to see whether your model works well, right inside of the playground. Note that the preceding code uses a `try!` statement. This is done to keep the code sample brief and simple. In your own apps, you should always strive for proper error handling to avoid surprising crashes.

The string that is passed to the `URL(fileURLWithPath:)` initializer, represents the location where your model will be stored. Make sure to specify the full path here, so, for instance, use `/Users/donnywals/Desktop/TextClassifier.mlmodel`, and not `~/Desktop/TextClassifier.mlmodel`.

The following lines of code test two different headlines to see if the model correctly labels them:

```
let techHeadline = try! model.prediction(from: "Snap users drop for  
first time, but revenue climbs")  
let politicsHeadline = try! model.prediction(from: "Spike Lee says  
President Donald Trump is a 'bullhorn' for racism")
```

If you're happy with the results of your model, you can grab the trained model from the place where you saved it, and immediately add it to your Xcode project. From there, you can use the model like you would use any other model.

Let's see another example of a Natural Language model. In this case, the model should label every word in a text to classify it as a certain type of word. For instance, you could train the model to recognize certain brand names, product names, or other words that have special meanings to your app. An example of training data that you could use to train a model like this is the following:

```
{  
    "tokens": ["Apple", "announced", "iOS 12", "and", "Xcode 10", "at",  
    "WWDC 2018"],  
    "labels": ["COMPANY", "NONE", "OPERATING_SYSTEM", "NONE",  
    "SOFTWARE", "NONE", "EVENT"]  
}
```

By collecting many samples that include the words that you want to label, your model will be able to not only match tags based on the word itself, but even on the surrounding words. Essentially, the model would be aware of the context in which each word is used to then determine the correct tag. Once you have collected enough sample data, you can train the model in a similar way as the classifier:

```
let labelTrainingData = try! MLDataTable(contentsOf:  
Bundle.main.url(forResource: "labels", withExtension: "json")!)  
let model = try! MLWordTagger(trainingData: labelTrainingData,  
tokenColumn: "tokens", labelColumn: "labels")  
try! model.write(to: URL(fileURLWithPath:  
"path/to/folder/TextTagger.mlmodel"))
```

The amount of code to train the model hasn't changed. The only difference is that the previous model was based on the `MLTextClassifier` class, and the current model is based on the `MLWordTagger`. Again, you can immediately use the trained model to make some predictions that you can then use to validate whether the model was trained properly. Providing good data and testing often are the keys to building a great CoreML model.

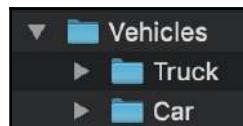
In addition to text analysis models, CreateML can also help you to train your own image recognition models. Let's see how this works next.

Training a Vision model

In the **ImageAnalyzer** sample app, you saw that picking an image of a certain car would be classified as a sports car with a pretty low confidence score. You can train your own vision model that specializes in recognizing certain cars.

Collecting good training data for image classifiers is tough, because you have to make sure that you gather many pictures of your subjects from all sides and in many different environments. For instance, if all your car images feature cars that are next to trees, or on the road, the model might end up classifying anything with trees or a road to be a car. The only way to obtain a perfect training set is to experiment, tweak, and test.

Training a Vision model works slightly different from training a Natural Language model. You can't use a JSON file to feed your test data to the classifier. So, instead, you should create folders that contain your images where the folder name is the label you want to apply to each image inside that folder. The following screenshot is an example of a training set that contains two kinds of labels:



Once you have collected your set of training data, you can store it anywhere on your computer—for instance, on the desktop. You will then pass the path for your training data to your model training code as follows:

```
import CreateML
import Foundation

let dataUrl = URL(fileURLWithPath: "/path/to/trainingdata")
let source = MLImageClassifier.DataSource.labeledDirectories(at:
dataUrl)
let classifier = try! MLImageClassifier(trainingData: source)
try! classifier.write(toFile: "~/Desktop/CarClassifier.mlmodel")
```

Again, you only need a couple of lines of code to train a model. That's how powerful CreateML is. If you want, you can quickly test your image classifier by dropping the `.mlmodel` file in the **ImageAnalyzer** project, and using that, instead of the MobileNet classifier that you used before.

Apart from the simple ways of training models, there are certain parameters that you can pass to the different CreateML classifiers. If you have trouble training your models properly, you could tweak some of the parameters that are used by CreateML. For instance, you could apply more iterations to your training set, so the model gains a deeper understanding of the training data.

As mentioned before in this chapter, machine learning is a subject that could span several books on its own, and even though CreateML makes training models straightforward and simple, it's not easy to train a robust model without any prior machine learning experience.

Summary

In this chapter, you have seen how you can make use of the machine learning capabilities that iOS provides. You saw that adding a machine learning model to your app is extremely simple, since you only have to drag it to Xcode and add it to your target app. You also learned how you can obtain models, and where to look to convert existing models to CoreML models. Creating a machine learning model is not simple, so it's great that Apple has made it so simple to implement machine learning by embedding trained models in your apps.

In addition to CoreML, you also learned about the Vision and Natural Language frameworks. Vision combines the power of CoreML and smart image analysis to create a compelling framework that can perform a massive amount of work on images. Convenient requests, such as facial landmark detection, text analysis, and more are available out of the box without adding any machine learning models to your app. If you do find that you need more power in the form of custom models, you now know how to use CreateML to train, export, and use your own custom, trained CoreML models. You learned that CreateML makes training models simple, but you also learned that the quality of your model is drastically impacted by the quality of your training data.

In the next chapter, you will learn how you can help your users live healthier lives with workouts and activity data.

Questions

1. What is the file format for a CoreML compatible model?
 - a) .coremlmodel
 - b) .mlmodel
 - c) .coreml
2. How do you use a CoreML model in your project?
 - a) By adding the model to Xcode and using the model class that is generated for you.
 - b) By adding the model to Xcode and training the model on the device.
 - c) By sending the CoreML model and the input data to iCloud.
3. Why is it so important the CoreML runs on the end user's device?
 - a) It's better for privacy and preserves the user's data plan.
 - b) Because the predictions can be calculated faster that way.
 - c) Both answers are true.
4. What does the NaturalLanguage framework do?
 - a) It generates natural text.
 - b) It extracts interesting information from text, for instance, names, places, words, and more.
 - c) It's used to determine whether a text was written by a human or a machine.
5. What does the Vision framework do?
 - a) It is used to access the user's camera.
 - b) It helps with accessibility for the visually impaired.
 - c) It performs complex analysis on images.
6. Which of the following options could negatively impact the quality of a training set for an image classifier?
 - a) If the training subject has different colors than the real object.
 - b) If the training subject is always shown in similar surroundings.
 - c) When the training subject is shown in many different angles. Natural Language

Further reading

- Apple's machine learning website is available at <https://developer.apple.com/machine-learning/>
- The CoreML tools documentation is available at <https://apple.github.io/coremltools/>
- The CreateML documentation can be found at https://developer.apple.com/documentation/create_ml
- The Natural Language framework's documentation is available at <https://developer.apple.com/documentation/naturallanguage>
- The Vision framework's documentation is available at <https://developer.apple.com/documentation/vision>

19

Tracking Activity Using HealthKit

More and more people care about their health nowadays. A lot of people try to make sure that they eat the right foods and maintain an active lifestyle. A considerable part of this group uses fitness apps and trackers, such as the Apple Watch, to measure and record their activities. All of this activity data is logged using Apple's HealthKit frameworks.

App developers can integrate HealthKit in their apps to track and record workouts, activities, and many other metrics that are meaningful for monitoring personal health. Because more and more people care about this data, Apple aims to continuously improve the HealthKit framework so that app developers can focus on building great apps, rather than figuring out how to integrate their data with iOS.

This chapter covers the following topics:

- Understanding HealthKit
- Implementing a workout app
- Integrating with iOS

By the end of this chapter, you will know how to build an application that tracks user activity in HealthKit. You will also know how to extract data from HealthKit so you can provide more accurate metrics for the workouts you log to HealthKit.

Understanding HealthKit

HealthKit is a framework that allows developers to interact with a user's health data. This data includes information like a user's weight, BMI, blood type, biological sex, skin type, date of birth, whether they use a wheelchair, and much more. In addition to these metrics, HealthKit also provides access to activity data, such as workouts or the amount of steps the user has walked on a given day.



HealthKit is available on the iPhone and Apple Watch only. The iPad does not have support for HealthKit, so you must take special precautions to determine whether HealthKit is available, by calling `isHealthDataAvailable()` on the `HKHealthStore` class.

Because health data is considered private and thus very sensitive, apps that want access to health data have to specify their reasons for wanting to read and/or write data to HealthKit in the `Info.plist`, under the following keys:

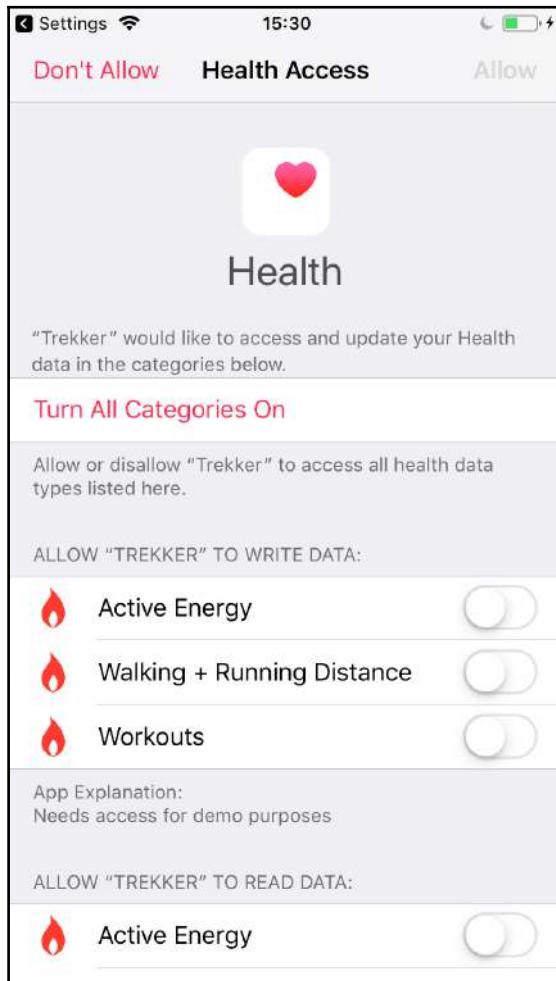
- Privacy: Health Share Usage Description
(`NSHealthShareUsageDescription`) for reading data from the HealthKit store
- Privacy: Health Update Usage Description
(`NSHealthUpdateUsageDescription`) for writing data to the HealthKit store
- Privacy: Health Records Usage Description
(`NSHealthClinicalHealthRecordsShareUsageDescription`) for reading record data from the HealthKit store

In addition to adding the appropriate `Info.plist` entries, you must also enable the HealthKit capability in your app's Capabilities tab. This will configure your app with the required entitlements for accessing the HealthKit store. When you enable the HealthKit capability, you have the option to enable access to health records. Health records are specific records that contain clinical data.

Requesting access to the HealthKit Store

Applications that want access to health-related data must ask permission for every type of data they want to access. If your app requires access to heart rate data, weight data, and activity data, that means that you must request access for each different data type. A single authorization request can ask for access to multiple data types at once, and the user can manually decide which data types can be accessed.

When you want to use certain data types from the HealthKit store, it's essential that you make sure your user understands why you need access to this data and to request access at an appropriate time. If your app has an onboarding flow, you could add a page that outlines the health data your app needs and asks the user for permission once they understand why you need access. The following screenshot shows the access dialog where users can configure the type of data an app is allowed to access:



The preceding screenshot shows all the different categories that this app wants to have read and write access for. An authorization request is performed to request access to a set of HealthKit categories. The following code is an example of the request that was used for the screenshot:

```
let objectTypes: Set<HKObjectType> = [HKObjectType.workoutType(),
HKObjectType.quantityType(forIdentifier: .activeEnergyBurned)!,  
HKObjectType.quantityType(forIdentifier: .distanceWalkingRunning)!]  
  
healthKitStore.requestAuthorization(toShare: objectTypes, read:  
objectTypes) { success, error in  
    print(success)  
    print(error)  
}
```

A user can revoke access to data at any time in the Health app, so apps must make sure that they presently have access to data before attempting to use it. The following code shows a sample of how to retrieve the authorization status for the `.distanceWalkingRunning` quantity type:

```
let objectType = HKObjectType.quantityType(forIdentifier:  
.distanceWalkingRunning)!  
let status = healthKitStore.authorizationStatus(for: objectType)
```

The possible values for the authorization status are the following:

- `.sharingAuthorized`
- `.sharingDenied`
- `.notDetermined`

Your app can't determine whether reading from the store is allowed. The reasoning for this is that denying an app access to data might hint that the user has something to hide. Since health data is very personal, it's important that apps can't make any assumptions based on the access they do or do not have the HealthKit. This means that if an app queries data while it doesn't have read access, HealthKit will return an empty result set. If the app has write access to HealthKit, only the results that the app wrote to the store will be returned.

Storing and retrieving data with HealthKit

All data that is stored in the HealthKit store is stored as an `HKSample` subclass, and is associated with a corresponding `HKObjectType` and an `HKObject`. Together, these objects describe what a user did, what value is associated with the sample, and when it occurred. The following code is a sample of adding a short walk of 100 meters to the HealthKit store:

```
let object = HKQuantity(unit: HKUnit.meter(), doubleValue: 100)
let objectType = HKObjectType.quantityType(forIdentifier:
    .distanceWalkingRunning)
let distance = HKQuantitySample(type: objectType!, quantity: object,
    start: Date(), end: Date())

healthKitStore.save(distance) { success, error in
    print(success)
    print(error)
}
```

To provide greater flexibility and unit conversion, HealthKit uses `HKUnit` objects to represent units like meters, inches, pounds, liters, and more. Because all HealthKit operations are asynchronous, the `save(_:_:withCompletion:)` method takes a completion callback that is called after the operation has completed.

To retrieve data from the HealthKit store, you use `HKQuery` objects. For instance, to retrieve a user's active energy burned for the past ten days, you would use a query that looks as follows:

```
let predicate = NSPredicate(format: "%K > %@",  
    HKPredicateKeyPathStartDate, (Date() - 60 * 60 * 24 * 10) as NSDate)
let sortDescriptors = [NSSortDescriptor(key:  
    HKSampleSortIdentifierStartDate, ascending: false)]
let objectType = HKObjectType.quantityType(forIdentifier:  
    .activeEnergyBurned) !
let query = HKSampleQuery(sampleType: objectType, predicate:  
    predicate, limit: Int(HKObjectQueryNoLimit), sortDescriptors:  
    sortDescriptors) { query, samples, error in
    print(samples)
    print(error)
    print(query)
}

healthKitStore.execute(query)
```

The preceding sample sets up an `NSPredicate` to filter the results that are returned by the query. HealthKit provides several helpers, including `HKPredicateKeyPathStartDate`, so you don't have to manually type the correct key paths that are used to filter and sort data.

The records that you receive from HealthKit contain metadata about the record itself. For instance, you can read the value that is associated with the record, the start and end dates, and even the source that added the record to the HealthKit store. All data that you retrieve from HealthKit is immutable. This means that you can't modify existing records that have already been saved to HealthKit.

The previous samples for storing and fetching data used metrics that are expressed as `HKQuantitySample` objects. This type of object is for data that is represented as a certain number. Walking distance, calories burned, or even weight are examples of this.

HealthKit also records data in the form of `HKCategorySample`. The value for a category sample is expressed as an enum value. Sleep tracking is an example of a metric that is tracked as a category sample. The user is either in bed, asleep, or awake. You can't add custom values to this type of sample.

Now that you are up to speed with gaining access to the HealthKit store and know how to read and write data, let's implement a small application that tracks a user's walks.

Implementing a workout app

In addition to measuring temporary individual metrics, you can track complete workouts with HealthKit. Workouts can contain data like the number of calories burned during the workout, the workout duration, the number of strokes that were made during a swim, and more. The most convenient way to track workouts is by adding a watchOS version of your app that uses all of the sensors in the Apple Watch to track a workout. Since this book focuses on iOS development, you won't learn how to track workouts using an Apple Watch app. Instead, you will implement a workout app that only uses the iPhone to track a running workout.

In the code bundle for this chapter, you will find a project called `Trekker`. This app is a straightforward run-tracking app, where users can tap a button to start tracking a workout. Their GPS location is tracked, and then entered into HealthKit when the workout is complete. The user's route will also be stored in HealthKit by implementing the `HKWorkoutRouteBuilder` class. This class takes GPS locations as they come in from a location manager and associates them with a workout.

The starter project for `Trekker` has already been set up with the appropriate permissions to begin implementing the HealthKit workout tracking immediately. The majority of your work should go on the `HealthKitHelper` class. This class is very similar to the `LocationHelper` that you worked on in [Chapter 17, Improving Apps With Location Services](#).

If you examine the `toggleWorkout()` method in `ViewController.swift`, you can already see how the app is supposed to work. If the health kit helper has all the required permissions and no workout is running, the HealthKit helper is asked to start a workout. At the same time, the location helper will begin tracking the user's location. One part that is still missing from this method is passing the locations received by the location helper on to the HealthKit helper.

When you associate locations with a workout route, it's important that the location data is as accurate as possible. So, before adding the location, you must make sure that the accuracy is good enough to use in a workout route. Add the following implementation for the `locationHelper.beginTrackingLocation(_:)` call:

```
locationHelper.beginTrackingLocation { [weak self] location in
    if location.horizontalAccuracy < 30 && location.verticalAccuracy <
    30 {
        self?.healthKitHelper.appendWorkoutData(location)
        self?.tableView.reloadData()
    }
}
```

When a location is added to the HealthKit helper, the table view is also reloaded so the user can see the tracked location data appear as it comes in.

With all the calls to the HealthKit helper in place, it's time to implement the missing methods to make sure the workout was tracked correctly.

There are four methods in `HealthKitHelper.swift` that need to be implemented to track the user's run and associate the relevant location data with it. The first step is to ask the user for permission to read and write their workout and workout route data. You need to ask for both of these permissions since you will store the workout and workout route data separately. Add the following implementation for `requestPermissions()` to ask for the correct permissions:

```
func requestPermissions() {
    guard isHealthKitAvailable
        else { return }

    let objectTypes: Set<HKSampleType> = [HKObjectType.workoutType(),
    HKSeriesType.workoutRoute()]
    healthKitStore.requestAuthorization(toShare: objectTypes, read:
    objectTypes) { success, error in
        if let error = error {
            print(error.localizedDescription)
        }
    }
}
```

If HealthKit is available on the current device, the user is prompted for access to workouts and workout routes. The next step is to implement `startWorkout()`. Add the following implementation to `HealthKitHelper`:

```
func startWorkout() {
    workoutBuilder = HKWorkoutRouteBuilder(healthStore: healthKitStore,
    device: nil)
    workoutStartDate = Date()
}
```

The preceding code creates a new instance of `HKWorkoutRouteBuilder` and associates it with the existing health store. You can optionally provide information about the device that is providing the route data. After creating the route builder, the current date is stored as the start date of the workout.

After creating an instance of `HKWorkoutRouteBuilder`, you can add new location data to it as the workout progresses. This location data is automatically attached to the route builder so it can eventually be connected to a workout and stored in HealthKit. Add the following implementation for `appendWorkoutData(_:)` to add new location data to the builder:

```
func appendWorkoutData(_ location: CLLocation) {
    workoutBuilder?.insertRouteData([location]) { success, error in
        if let error = error {
            print(error.localizedDescription)
    }
}
```

```
        } else {
            print("Location data added to route")
        }
    }
}
```

The preceding code calls `insertRouteData(_:completion:)` to add new data to the builder. Since this operation is performed asynchronously, this method takes a completion callback. In this case, the result isn't very relevant, so the result is only printed to the console.

There is one more step left to wrap up the workout tracker. When `endWorkout()` is called, an instance of `HKWorkout` should be created and stored to HealthKit. After saving the workout, the route should be associated with this workout and saved to HealthKit. Calling `finishRoute(with:metadata:completion:)` on the route builder will associate it with the workout and store it in HealthKit.

Add the following implementation for `endWorkout()` to the `HealthKitHelper` class:

```
func endWorkout() {
    guard let builder = workoutBuilder, let startDate = workoutStartDate
    else { return }

    let workout = HKWorkout(activityType: .running, start: startDate,
                           end: Date())

    healthKitStore.save(workout) { success, error in
        builder.finishRoute(with: workout, metadata: [ : ]) { route, error
            in
                if let error = error {
                    print(error.localizedDescription)
                } else {
                    print("route saved: \(route.debugDescription)")
                }
            }
        }
    }
}
```

This final method implementation wraps up the implementation of your custom workout tracker. Go ahead, launch your app and have a short walk outside to see your walk being logged right into your health app.

Summary

In this chapter, you learned about HealthKit. You learned what it is, how to ask the user for permissions, and how you can read and write data in the HealthKit store. This chapter put a lot of focus on using HealthKit to track a user's activity, but HealthKit is capable of tracking much more data. You can build apps that help users track just about anything related to their health.

When you build an app that accesses a user's health data, make sure that you don't request access to information that you don't need. It's usually better to ask the user for permission to access certain data just before you need it than to ask for all permissions at once. For instance, you might build an app that tracks both sleep and blood sugar, but not necessarily at the same time. You might want to ask permission to access each of these data types separately, so the user fully understands why you need access to their data.

In addition to learning how you read and write HealthKit data, you implemented a straightforward workout app that can track a user while they are walking around outside, and associate that with a workout which is then saved to HealthKit. With the knowledge from this chapter, you should have an excellent starting point from which you can start building your own HealthKit-based apps to help your users live a healthier life.

In the next chapter, you will learn how you can use Siri and the new Siri Shortcuts to make your user's life a little bit easier.

Questions

1. Which devices support HealthKit?
 - a) The iPhone and iPad.
 - b) The iPhone, iPad, and Apple Watch.
 - c) The iPhone and Apple Watch.

2. What's the difference between a quantity sample and a category sample?
 - a) Quantity samples have an arbitrary value, while categories use enums to represent their value.
 - b) Quantity samples take place over time, and category samples are static.
 - c) Category samples have more possibilities than quantity samples.
3. How can you check whether you have read access to a certain health metric?
 - a) By requesting the authorization status for that metric.
 - b) By making a request. If the results are empty, you don't have access.
 - c) To protect the user's privacy, there is no way to be sure.
4. How can you filter data that is returned from a HealthKit query?
 - a) With predicates
 - b) By using the built-in filters
 - c) By specifying the values you want to fetch
5. What permissions do you need to use a workout route builder?
 - a) Only for workouts.
 - b) Just for the workout route builder.
 - c) You need permission for both the builder and the workouts.
6. What do you need to keep in mind when you associate location data with a workout route?
 - a) You should not add too much data.
 - b) You should make sure the location data is very accurate.
 - c) You should make sure to add all location data.

Further reading

- Apple's HealthKit documentation is available at <https://developer.apple.com/documentation/healthkit>

20

Streamlining Experiences with Siri

Over the past few years, Apple has provided developers with more and more opportunities to integrate their apps with iOS. One of these opportunities is in integrating your application with Siri using the `SiriKit` framework.

In this chapter, you will work on an existing application called `Hairdressers`. This application allows users to book an appointment with their hairdresser for a certain day, and can also message their hairdresser directly. You will learn how you can integrate Siri Intents extensions to allow users to message their hairdresser through Siri. You will also implement the new Siri shortcuts to allow users to quickly schedule their appointments with their regular hairdresser.

In this chapter, you will learn about the following topics:

- Understanding intents
- Implementing an intents extension
- Adding a custom UI to a Siri intents extension
- Integrating Siri Shortcuts in your app

If you want to follow along with the code samples in this chapter, make sure to grab the `Hairdressers` starter project from this book's code bundle. Make sure to examine the existing code, so you understand what's going on inside of the app. The code relies heavily on the knowledge you gained in [Chapter 12, Adding Core Data to Your App](#), so you might want to keep that chapter nearby while you examine the starter code.

Understanding intents

Siri is a powerful, smart, and ever-learning personal assistant that aims to give natural responses to natural speech input. This means that there is often more than one way to say something to Siri. Some users like to be extremely polite when they talk to Siri, saying please and thank you whenever they ask for something. Other users like to be short and to the point; they simply tell Siri what they want and that's it.

This means that Siri has to be really smart about how it interprets language and how it converts the user's requests to actionable items. Not only does Siri take into account the language used; it's also aware of how a user is using Siri. If a user activates Siri by saying *Hey, Siri!*, Siri will be more vocal and verbose than when a user activates Siri by pressing and holding the home or side buttons because it's more likely if they didn't press the home or side button, that the user is not looking at their device.

To convert a user's spoken requests into actions, Siri uses intents. An **intent** is a way to describe an action. These intents are supported by app-specific vocabularies; this allows you to make sure that your users can use terms that are familiar to them because they are also used in your app.

Siri does not handle an unlimited amount of intents. All of the intents that Siri can handle belong to a predefined set of domains. If you plan to create a Siri extension for your own app, it must fit into one of the predefined domains that Siri understands—unless you provide a custom intent. Currently, Siri handles the following domains out of the box:

- Messaging (*Send a message to Donny that says Hello, World*)
- Calling (*Call my sister*)
- Payments (*Transfer 5 euros to Jeff*)
- Workouts (*Start an outdoor run*)
- Ride booking (*Get me a taxi to the airport*)
- Photo search (*Find me photos from Paris*)
- Notes (*Create a note that says "Don't forget the dishes"*)
- Reminders (*Remind me to do the dishes when I get home*)
- Visual codes (*Show me my ticket for tonight's concert*)

If your app is not in one of these domains, you will need to do a bit more work to integrate your app with Siri, as you will learn in the last section of this chapter. In order to integrate Siri with your app, your app must ask the user permission for it to be used with Siri. It's recommended that you do this when your user first opens your app because this ensures that your app is available to Siri as soon as possible. Make sure that you ask for permission appropriately, though; you don't want to unpleasantly surprise your user with a permission request. You also need to add the *Privacy—Siri Usage Description* (`NSSiriUsageDescription`) key to your `Info.plist` file, and enable the Siri capability in your app's Capabilities tab.



You can only enable the Siri capability for your app if you are a paying member of Apple's Developer Program. Unfortunately, you can't enable or test Siri if you haven't got such a membership.

In order for Siri to work with your app, it's important that a user mentions your app's name to Siri when asking it something related to your app. This is to make sure that apps don't hijack certain words or verbs, which would result in a confusing experience. It doesn't matter exactly how or when the user does this. Siri is all about natural language, and it will even understand if your app name is used as a verb. For example, *MyBankApp some money to John Doe* would be interpreted as a money transfer action with an app named *MyBankApp* to a person named *John Doe*.

If a user would literally ask Siri to perform the preceding action, Siri would not be able to fulfill the request yet. In the case of a money transfer Siri has to know two things:

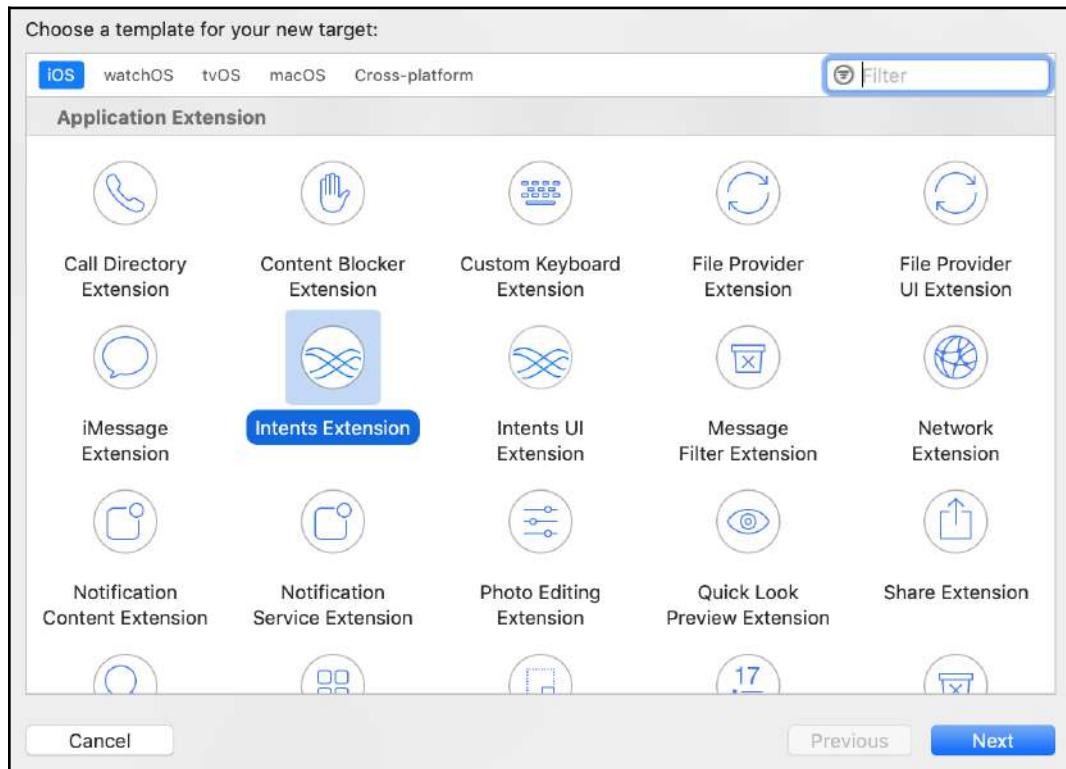
1. Who should receive the money?
2. How much money should be transferred?

These requirements are captured in an intent, and if Siri is missing information, it will ask the user directly for this information in order to collect everything needed to successfully transfer the money.

Now that you know what an intent is and what it's used for, let's see how you can implement an intents extension so Siri knows how to use your app to send messages.

Implementing an intents extension

There are several ways for apps to integrate with iOS. These integrations are always built using app extensions. An app extension is an isolated section of your app that can be instantiated and used by the system as needed to provide certain functionality for users. In iOS 12, there are 25 different ways for apps to integrate with iOS. Not every app can or should implement every possible extension, but a proper integration with iOS can truly make your app stand out. The following screenshot shows a selection of available extensions in iOS 12:

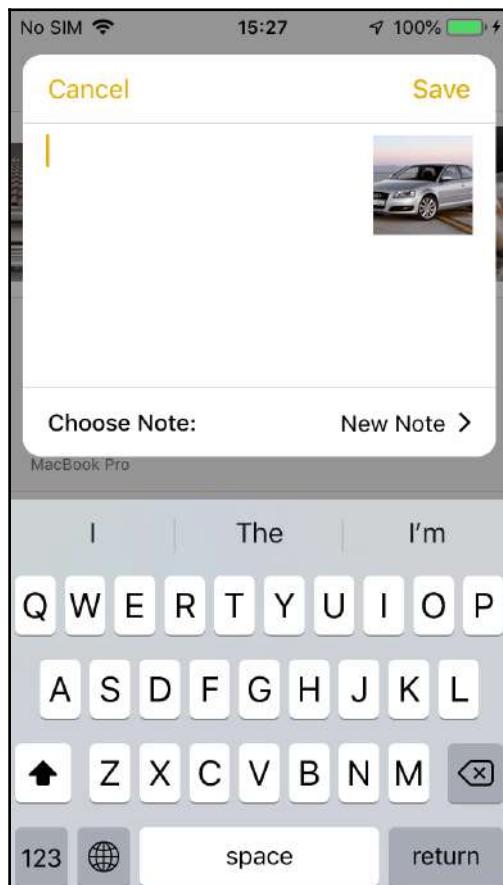


In this chapter, you will implement an **Intents Extension** and an **IntentUI Extension**. These extensions are used to communicate with Siri and to provide a custom UI for the Siri integration. Before you implement your extensions, let's see what extensions are and how they work.

Understanding app extensions

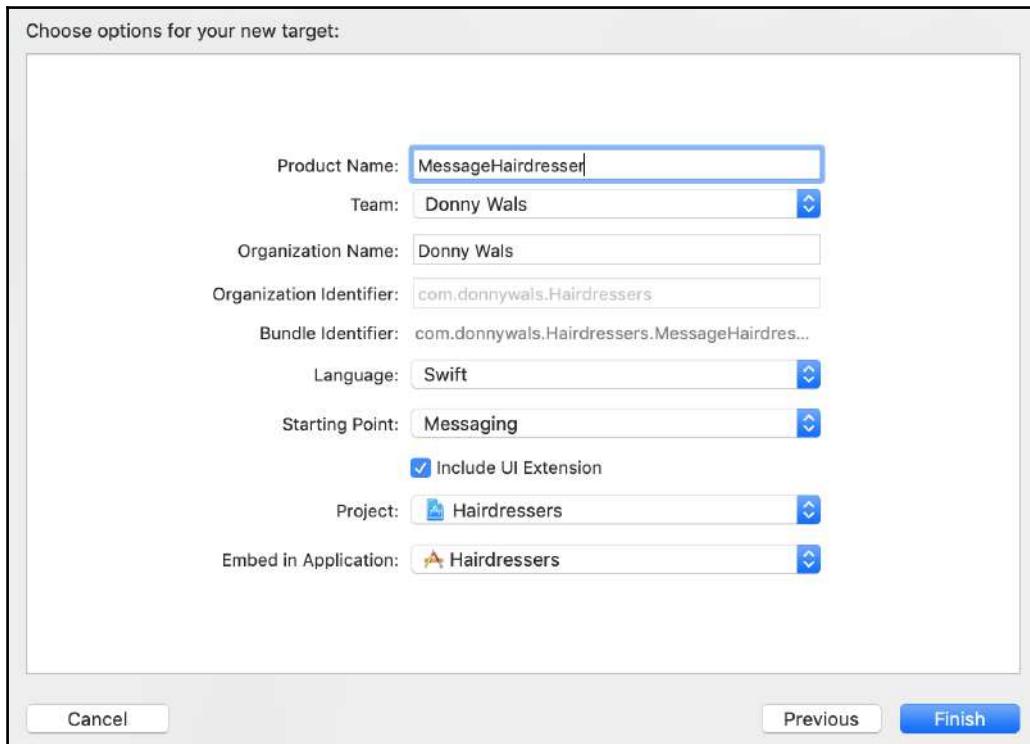
As mentioned before, extensions are used by apps to integrate with iOS. Some extensions have their own UI, such as the **IntentUI Extension**, and others have no UI at all, such as the **Intents Extension**. Every extension has its own purpose and specializes in performing a single task.

When an extension is opened or instantiated, this happens through the extension framework itself. For instance, if you open a photo in the **Photos** app and you tap the share option, you can choose several to choose the selected photo too. If you select one of these apps, its **Share Extension** is launched. This extension then provides its own UI that is used to share the photo in a way that is supported by the share target. The following screenshot is an example of this:



In the preceding screenshot, a photo is shared to the **Notes** app. The Notes app itself does not launch or own the extension. This means that an extension functions completely separately from its host application. Instead, the extensions framework is responsible for managing the extension's life cycle. Because of this, extensions do not work like individual apps. Extensions usually only have a single view controller, and you have to rely on the `UIViewController` life cycle methods to configure your extension. This might seem a little bit counter-intuitive at first because, as you'll find out in a moment, extensions have their own build target and `Info.plist`, which makes them look a lot like a separate app, while they behave more like a view controller.

Extensions are added to an app through the **Project** window. If you look at the sidebar in the center window, you can see your project name and the **Targets** associated with your project. Each of these targets produces its own build product. The app target produces your app itself, for instance. You can use the + icon at the bottom of the sidebar to add a new target to your app. Go ahead and add an extension for the Hairdressers app. Make sure to select the **Intents Extension** option and check the **Include UIExtension** checkbox as shown in the following screenshot:



Now that you have an understanding of app extensions, let's see how you can implement your own **Intents Extension** to integrate the Hairdressers app with Siri.

Configuring your extension

Now that you have added your extensions to the Hairdressers project, it's time to configure the app so it can integrate with Siri. Go through the following steps to properly configure your app:

- Make sure you're using a paid developer team, and make sure the app uses this paid team by selecting it in the project settings.
- Enable the Siri capability in the **Capabilities** tab.
- Add the required Privacy key to the app's `Info.plist` file.

After configuring the project, the first step to integrate your app with Siri is to ask the user for permission to do so. Open `AppointmentsViewController.swift` and import the `Intents` framework at the top of the file as follows:

```
import Intents
```

You might have expected all of the Siri-related code to exist in the `SiriKit` framework. Instead, some parts of the Siri integration live in the `Intents` framework, because some other apps, like the `Maps` app on iOS, use intents to determine what action a user wants to perform.

After adding the import, add the following implementation for `viewDidAppear(_ :)` to `AppointmentsViewController`:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    INPreferences.requestSiriAuthorization { status in
        print(status == .authorized)
    }
}
```

This preceding code asks for permission to use Siri in this app as soon as possible, just like Apple recommends. If permission is already granted, the permissions dialog won't be displayed, so there is no need to check for the current status prior to asking permission. For now, you have done all the required work in the app.

If you open the `MessageHairdresserIntent` folder that Xcode created when you added the extension earlier, you should see that there are two files in there: a file named `IntentHandler.swift`, and the app extension's `Info.plist`. The `IntentHandler` class is responsible for communicating with Siri to resolve, confirm, and handle the intents for which the extension is registered. The `Info.plist` is used to determine which intents can be handled by the extension.

Open the `Info.plist` file and expand the `NSExtension` key. You'll notice that there are two intent-related keys in the file: `IntentsRestrictedWhileLocked` and `IntentsSupported`. The second key contains all of the intents that can be handled by the extension. The `IntentsRestrictedWhileLocked` key specifies which of these supported keys can or can't be used without unlocking the device. SiriKit itself will lock some intents by default. Money transfers, for example, can't be done without unlocking the device, regardless of your extension settings:

▼ NSExtension		
Dictionary (3 items)		
▼ NSExtensionAttributes	Dictionary	(2 items)
▼ IntentsRestrictedWhileLocked	Array	(0 items)
▼ IntentsSupported	Array	(3 items)
Item 0	String	INSendMessageIntent
Item 1	String	INSearchForMessagesIntent
Item 2	String	INSetMessageAttributeIntent

The list of intents in `IntentsSupported` is a list of intent class names that your extension is able to handle. Xcode has added a couple of example intents, but this list is not even close to being an exhaustive list of available intents. For a complete list of available intents, you should have a look at the documentation for the `Intents` framework.

The available intents range from starting a workout, to booking a restaurant reservation or requesting that another person transfers money into your bank account. Each of these intents has their own corresponding class that holds all of the properties that are used to describe the intent.

To enable message sending in Hairdressers, only the `INSendMessageIntent` is required, so you can remove the other two intents. If you want to experiment with multiple intents from the get-go, you can keep any intents that you want to play around with, or add more, if you like!

Even though Siri is quite smart when it comes to resolving intents, some apps have their own terminology for specific actions. Custom terms make resolving intents a lot harder for Siri. Luckily, you can help Siri out by adding vocabulary information to Siri.

Adding vocabularies to your app

Siri always makes an effort to understand what a user is trying to do with an app and is usually quite good at this. However, sometimes it's tough for Siri to figure out what's going on or what a user is trying to do. To help Siri figure out what a user might mean, you can provide vocabularies. A **vocabulary** is a set of strings that map to intents, or to parameters for the intent.

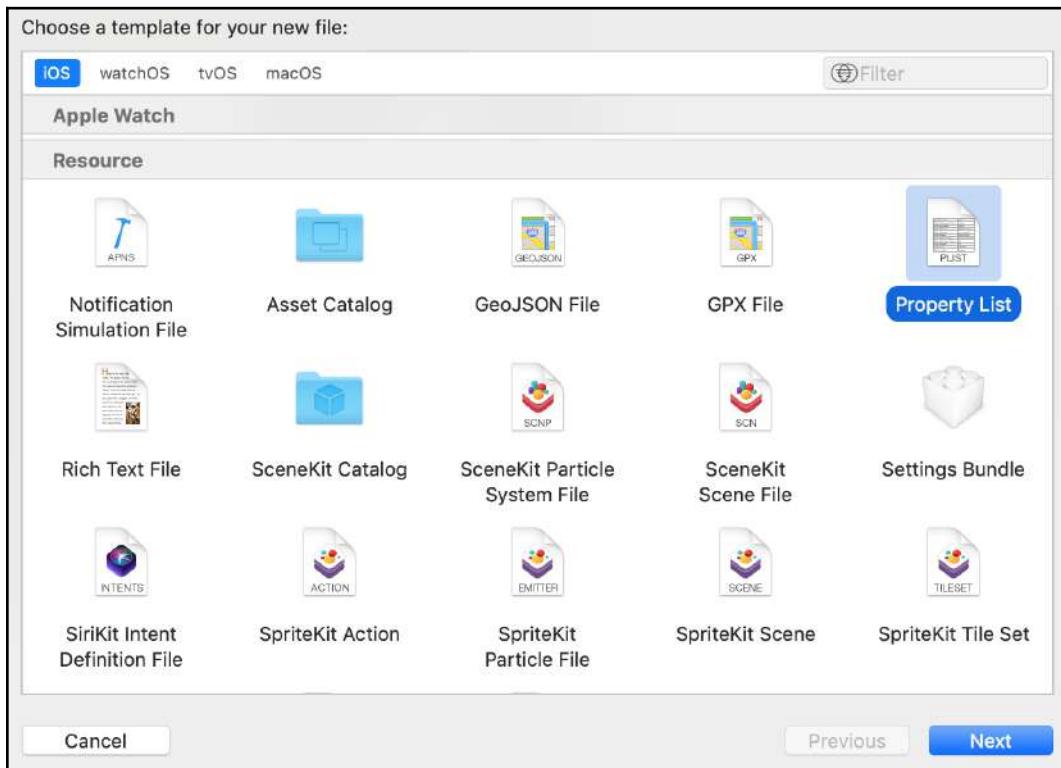
There are two ways for you to teach Siri the vocabulary for your app. One way is through a `.plist` file. This approach is mostly used for when your app has a global vocabulary that applies to all users. If your app can't supply all custom vocabularies in a static file, you can provide them dynamically. This is particularly useful if, for instance, you want to teach Siri about contacts in a messaging app.

Adding vocabularies through a `.plist` file

You already learned that Siri understands when a user wants to do something with your application, even if your user uses your app name as a verb. However, your app name might not be the only thing that's specific to your app. Let's see how this works for a workout app.

If, for example, a user were to tell Siri, *Hey Siri, start an Ultimate Run using RunPotato*, Siri would be able to figure out what RunPotato is since it's the app that is expected to handle the intent. What it won't be able to understand instantly is what it means to *start an Ultimate Run*. This is where a custom vocabulary entry in a `.plist` makes a lot of sense.

The .plist that contains the vocabulary should be provided by the application itself; the corresponding extension is not able to do this. You can add a vocabulary of app-specific words, like the workout name above, by adding an extra .plist file to the app. To add a new .plist file, add a new file to your app target and select the **Property List** file type under the **Resource** header as shown in the following screenshot:



The file you create must be named `AppIntentVocabulary`. This file contains all of the information about an app's custom vocabulary. The vocabulary .plist is pretty specific and verbose. This is important because its purpose is to provide a window into the vocabulary your app uses. When you created the `AppIntentVocabulary.plist` file, Xcode added a dictionary at the top level of the file. All custom vocabulary should be added to that dictionary.

To implement the example phrase shown earlier, you need a `Parameter Vocabularies` array. This array will contain `Parameter Vocabulary` items that specify the vocabulary the app uses for workout names. The first item that needs to be defined is a `Parameter Names` array. For the workout app, it would contain a single entry: `INStartWorkoutIntent.workoutName`. In addition to the parameter names, a `Parameter Vocabulary` dictionary must also be added to the entry in the `Parameter Vocabularies` array.

The `Parameter Vocabulary` dictionary has keys for the item identifier, synonyms for this identifier, a pronunciation, and even an example phrase. This is all very verbose, but it provides Siri with all of the information it needs to resolve user input for your app. The following screenshot shows an example of the entry that would be needed to make sure Siri understands what an *Ultimate Run* workout is:

▼ App Intent Vocabulary Property List	Dictionary	(1 item)
▼ Parameter Vocabularies	Array	(1 item)
▼ Item 0 (Parameter Vocabulary)	Dictionary	(2 items)
▼ Parameter Names	Array	(1 item)
Item 0 (Parameter Name)	String	<code>INStartWorkoutIntent.workoutName</code>
▼ Parameter Vocabulary	Array	(1 item)
▼ Item 0 (Parameter Vocabulary)	Dictionary	(2 items)
Vocabulary Item Identifier	String	<code>ultimate run</code>
▼ Vocabulary Item Synonyms	Array	(1 item)
▼ Item 0 (Vocabulary Item Synonym)	Dictionary	(3 items)
▼ Vocabulary Item Examples	Array	(1 item)
Item 0 (Vocabulary Item Example)	String	<code>Start an Ultimate Run with RunPotato</code>
Vocabulary Item Phrase	String	<code>Ultimate Run</code>
Vocabulary Item Pronunciation	String	<code>ultemt run</code>

Teaching Siri new vocabularies at runtime

The second way you can teach Siri about specific content in your app is through the `INVocabulary` class. The `INVocabulary` class is used to teach Siri information that is specific to the user or changes over time.

A great example of user-specific vocabulary is a workout program that your user manually creates inside of your app. Or maybe the user's friends in a messaging app, if those friends don't match the people your user has in their list of contacts on their device.

Updating the vocabularies for an app always occurs in batches. You don't add or remove individual words or phrases for a specific string type. For example, if your user adds a new workout type to a workout application, or if they add new content in a messaging application, you must update the entire vocabulary for the parameter that the vocabulary applies to. Since the Hairdressers app uses its own list of hairdressers, Siri should learn their names so the users can message them. Add the following code to the application (`_:didFinishLaunchingWithOptions:`) method in `AppDelegate`, and don't forget to import the `Intents` framework, as follows:

```
let hairdressers = NSOrderedSet(array:  
    HairdressersDataSource.hairdressers)  
INVocabulary.shared().setVocabularyStrings(hairdressers, of:  
    .contactName)
```

By supplying the contact names vocabulary when the app starts, it's available to Siri as soon as possible, even before the app's first screen is loaded.

If a user logs out or does something else that makes their custom vocabulary redundant or invalid, you need to make sure you delete the entire user-specific vocabulary by calling `removeAllVocabularyStrings()` on the shared `INVocabulary` class. This method of teaching vocabulary to Siri is not intended for vocabularies that are common to all users. If a vocabulary is the same for all users of your app, you should provide this vocabulary through the `.plist` file that was mentioned earlier. Now that you're completely up to speed regarding intents and vocabularies, let's see how you can start to handle the intents that Siri sends your way.

Handling intents in your extension

Handling intents can be divided into three stages. The first stage is the resolving stage. In this stage, your extension will go back and forth with Siri to figure out the correct parameters for the given intent. In this stage, Siri could ask your app to verify that a certain username exists. Your extension will then have to figure out if the given input is valid, and you'll provide Siri with a response code that tells it whether the parameter is resolved, or maybe requires a little bit more clarification on the user's end.

The second stage is expected to confirm that everything is set up correctly and all requirements for executing the action are met. The third and final stage involves actually acting on the intent and performing the desired action. Let's go through each of stages, one by one.

Resolving the user's input

When you create an intents extension, Xcode creates a single main class for your extension, named `IntentsExtension`. This is the class that serves as an entry point for your extension. It contains a `handler(for:)` method that returns an instance of `Any`. The `Any` type indicates that this method can return virtually anything and the compiler will consider it valid. Whenever you see a method signature like this, you should consider yourself on your own. Being on your own means that the Swift compiler will not help you to validate that you've returned the correct type of object from this method.

In addition to the `handler(for:)` method, Xcode has generated a lot of sample code to show how you could implement an intent handler. After quickly studying this code, you can go ahead and remove it, since you'll write your own implementation for this functionality soon.

The reason the `handler(for:)` method returns `Any` is because this method is supposed to return a handler for every intent that your app supports. If you're handling a send message intent, the handler is expected to conform to the `INSendMessageIntentHandling` protocol. Xcode's default implementation returns `self`, and the `IntentHandler` class conforms to all of the intents the extension handles by default, according to its plist file.

This default approach is not inherently wrong, but if you add an intent to your extension and forget to implement a handler method, you might return an invalid object from the `handler(for:)` method. A cleaner approach is to check the type of intent you're expected to handle, and return an instance of a class that's specialized to handle the intent. This is more maintainable, and will allow for a cleaner implementation of both the intent handler itself, and the `IntentHandler` class.

Replacing Xcode's default implementation ensures that you always return the correct object for every intent that the Hairdressers app supports. Go ahead and replace the `handler(for:)` method with the following implementation:

```
override func handler(for intent: INIntent) -> Any? {
    if intent is INSendMessageIntent {
        return SendMessageIntentHandler()
    }

    return nil
}
```

The `SendMessageIntentHandler` is a class you will define and implement to handle the sending of messages. Create a new `NSObject` subclass, named `SendMessageIntentHandler`, and make it conform to `INSendMessageIntentHandling`. When you create this file, make sure it's added to the **MessageHairdresserIntent** target, and not the Hairdressers app target.

Every intent handler has different required and recommended methods. `INSendMessageIntentHandling` has just one required method: `handle(sendMessage:completion:)`. Other methods are used to confirm and resolve the intent. All of the resolve methods work in similar ways, but are used for different parameters in an intent.

Imagine you're building a messaging app that uses groups to send a message to multiple contacts at once. These groups are defined in your app and Siri wants to resolve a group name. If this occurs, Siri calls the `resolveGroupName(forSendMessage:with:)` method on the intent handler. This method is now expected to resolve the group name and inform Siri about the result by calling the callback it's been passed. Let's see how this works:

```
let supportedGroups = ["neighbors", "coworkers", "developers"]

func resolveGroupName(forSendMessage intent: INSendMessageIntent, with completion: @escaping (INStringResolutionResult) -> Void) {

    guard let givenGroupName = intent.speakableGroupName else {
        completion(.needsValue())
        return
    }

    let matchingGroups = supportedGroups.filter { group in
        return group.contains(givenGroupName)
    }
```

```
switch matchingGroups.count {  
    case 0:  
        completion(.needsValue())  
    case 1:  
        completion(.success(with: matchingGroups.first!))  
    default:  
        completion(.disambiguation(with: matchingGroups))  
    }  
}
```

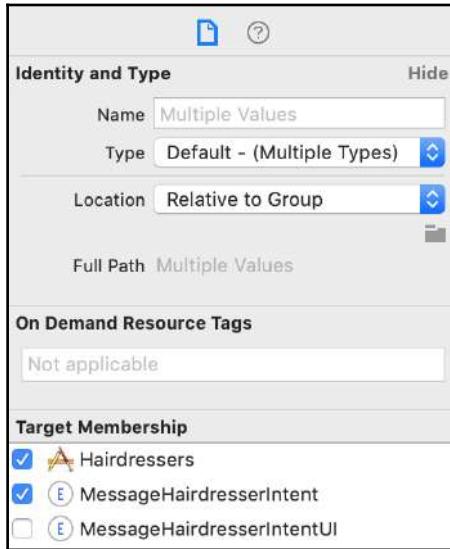
To simplify the example a bit, the supported groups are defined as an array. In reality, you would use the given group name as input for a search query in Core Data, your server, or any other place where you might have stored the information about contact groups.

The method itself first makes sure that a group name is present on the intent. If it's not, a callback is used to inform Siri that a group name is required for this app. Note that this might not be desirable for all messaging apps. Actually, most messaging apps will allow users to omit the group name altogether. If this is the case, you'd call the completion handler with a successful result.

If a group name is given, it is used to filter the `supportedGroups` array. Again, most apps would query an actual database at this point. If no results are found, Siri is asked for a value. If a single result is found, the work is done. The code successfully managed to match the intent's group with a group in the app's database and Siri is informed accordingly. If more than one result was found, Siri is asked to disambiguate the results that were found. Siri will then take care of asking the user to specify which one of the provided inputs should be used to send the message to. This could happen if you ask Siri to send a message to a person named *Jeff*, but you have multiple *Jeffs* in your contact list.

In the case of the Hairdressers app, messages are sent to individual hairdressers that are stored in the `Hairdressers.plist` file. The `HairdressersDataSource` helper object can read the data from this `plist` and provides a simple array of hairdresser names. Since this data is currently only part of the Hairdressers app target, you will need to add it to the `MessageHairdresserIntent` target as well.

To do this, select both of the files in the **Project Navigator**, and use the **File Inspector** on the right side of the Xcode window to add these files to the **MessageHairdresserIntent** target, as shown in the following screenshot:



Next, add the following extension to `SendMessageIntentHandler.swift` to implement `resolveRecipients(for:with:)`:

```
extension SendMessageIntentHandler: INSendMessageIntentHandling {
    func resolveRecipients(for intent: INSendMessageIntent, with completion: @escaping ([INSendMessageRecipientResolutionResult]) -> Void) {

        guard let recipients = intent.recipients else {
            completion([.needsValue()])
            return
        }

        let results: [INSendMessageRecipientResolutionResult] =
        recipients.map { person in
            let matches = HairdressersDataSource.hairdressers.filter { hairdresser in
                return hairdresser == person.displayName
            }

            switch matches.count {
            case 0: return
            INSendMessageRecipientResolutionResult.needsValue()
        }
    }
}
```

```
        case 1: return
    INSendMessageRecipientResolutionResult.success(with: person)
        default: return
    INSendMessageRecipientResolutionResult.disambiguation(with: [person])
    }
}

completion(results)
}
}
```

The preceding code should look very familiar, because it's very similar to the code for resolving a group name. The main difference is that a user can choose multiple recipients for the message, so a resolution result is created for each of the recipients that Siri passes to the `resolveRecipients(for:with:)` method. The next stage in handling the intent is to confirm the intent status.

Confirming the intent status

After you've made sure that everything is in place to eventually handle the intent, you must confirm this to Siri. Every intent handler has a `confirm` method. The signature might vary, but there is always some form of confirmation. Refer to the documentation for the intent you are handling to confirm which method you are expected to implement. When you send messages, the confirmation method is `confirm(sendMessage:completion:)`.

You can make the confirmation step as complex as you want. For example, you could check whether a message is too long, contains forbidden content, or virtually anything else. Most commonly, you'll want to make sure that the user is authenticated and allowed to send a message to the recipient.

Again, it's completely up to your best judgment to determine which conditions apply to your extension. The important takeaway for the `confirm` method is that you're expected to make sure that everything is in place to smoothly perform the action later.

Let's look at an example of a confirmation implementation to explore some of the possible outcomes of the confirmation stage:

```
func confirm(sendMessage intent: INSendMessageIntent, completion:
@escaping (INSendMessageIntentResponse) -> Void) {

    guard let user = User.current(), user.isLoggedIn else {
        completion(INSendMessageIntentResponse(code:
.failureRequiringAppLaunch, userActivity: nil))
    }
}
```

```
        return
    }

    guard MessagingApi.isAvailable else {
        completion(INSendMessageIntentResponse(code:
.failureMessageServiceNotAvailable, userActivity: nil))
        return
    }

    completion(INSendMessageIntentResponse(code: .ready, userActivity:
nil))
}
```

The preceding implementation checks whether a current user is available, and whether they are logged in or not. If no user exists, the intent handler will inform Siri that the app should be launched so that a user can log in.

Next, the availability of the API that will eventually handle the message sending is checked. Note that the `user` and `api` classes don't exist in the example project, and should be defined by you if you decide to go with this confirmation approach. These classes simply serve as placeholder examples to demonstrate how confirmation of an intent works.

If a user must be taken to your app in order to log in or for any other reason, Siri will automatically create a user activity that's passed to `AppDelegate` in your application. You must implement `application(_:continue:restorationHandler:)` to catch and continue the user activity.

A user activity that's created by Siri has its `interaction` property set. This property contains an `INInteraction` object that reflects the action that the user attempts to complete using Siri. A good implementation of `application(_:continue:restorationHandler:)` will fulfill this interaction as soon as possible inside of the app. It's also possible to create your own user activity if you want to add custom information that Siri doesn't pass on. If you want to do this, you should pass your custom user activity to the `INSendMessageIntentResponse` initializer.

Since the Hairdressers app doesn't use an external API and the user doesn't have to log in, the `confirm(sendMessage:completion:)` method does not have to be implemented. After ensuring that everything is in place, it's time to move on to the third stage of handling an intent: performing the desired action.

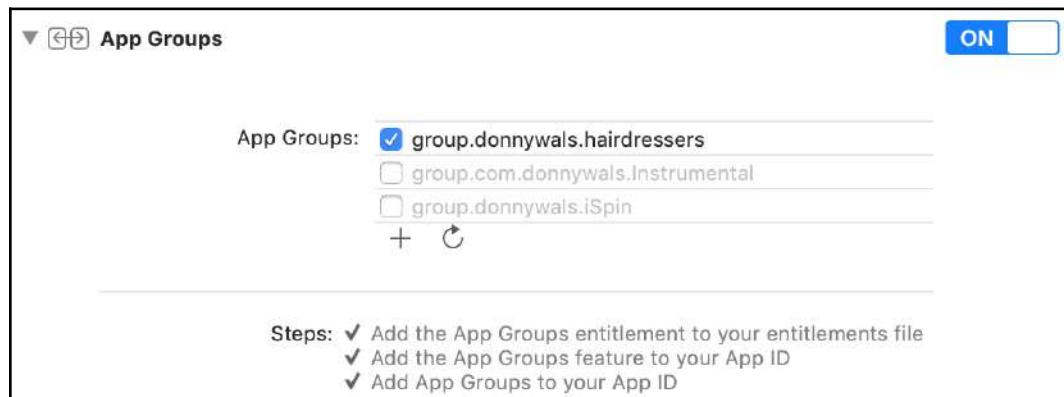
Performing the desired action

Once Siri understands what the user wants to do and which parameters to use, and once your app has confirmed that everything is in place to handle the user's request, the time has finally come to execute the requested action. Once this time has come, Siri calls the `handle` method on your intent handler.

Just like the `confirm` method, every intent has their own version of this method, but they all follow a similar pattern. For sending messages, the method signature is `handle(intent:completion:)`, where the `intent` is an instance of `INSendMessageIntent`. The parameters for this method are identical to the ones in the confirmation step. The major difference is that you're now expected to handle the intent, instead of only confirming that the intent is valid.

Once you have handled the intent, you must call the completion handler with an `INSendMessageIntentResponse`. If everything goes well, you're expected to use a `success` response code. If you're unable to process the intent promptly, you must call the completion handler with an `inProgress` status code. Using the `inProgress` status code informs Siri that you're handling the intent, but it's taking a while. If you fail to handle the intent at all, you should pass a `failure` status to the completion handler.

Since the Hairdressers app uses Core Data to store all of its data, you will need to perform some additional steps to share data between the extension and the app. To do this, you must use the **App Groups** capability. **App Groups** associate multiple targets with a single, shared group, which allows them to access shared files or `UserDefault`s instances. To create an app group, you must enable the **App Groups** capability in the project's **Capabilities** tab, and provide a name for your group, as shown in the following screenshot:



When you have configured the app groups capability for your app, enable it for the **MessageHairdresserIntent** extension as well, and make sure to select the app group you just created to add the extension to that app group.

After doing this, you must configure the app's persistent container to use the app group for storing the database file. Add the following code to the `persistentContainer` computed property in `PersistentHelper`, right before the persistent store is loaded, to configure the persistent container properly:

```
let containerUrl =
    FileManager.default.containerURL(forSecurityApplicationGroupIdentifier
        : "YOUR.APP.GROUP.NAME")!
let databaseUrl =
    containerUrl.appendingPathComponent("Hairdressers.sqlite")
let description = NSPersistentStoreDescription(url: databaseUrl)
container.persistentStoreDescriptions = [description]
```

Also, make sure to add the `PersistentHelper.swift` file, the `NSManagedObjectContext` extension, and the data model in `Hairdressers.xcdatamodeld`, which already exist in the app target, to the extension target.

Now that the app and its Siri extension share the Core Data store, you are ready to take the message the user is trying to send and store it in the Core Data database. Add the following code to the existing extension on `SendMessageIntentHandler`:

```
func handle(intent: INSendMessageIntent, completion: @escaping
    (INSendMessageIntentResponse) -> Void) {
    guard let hairdressers = intent.recipients, let content =
        intent.content else {
        completion(INSendMessageIntentResponse(code: .failure,
            userActivity: nil))
        return
    }

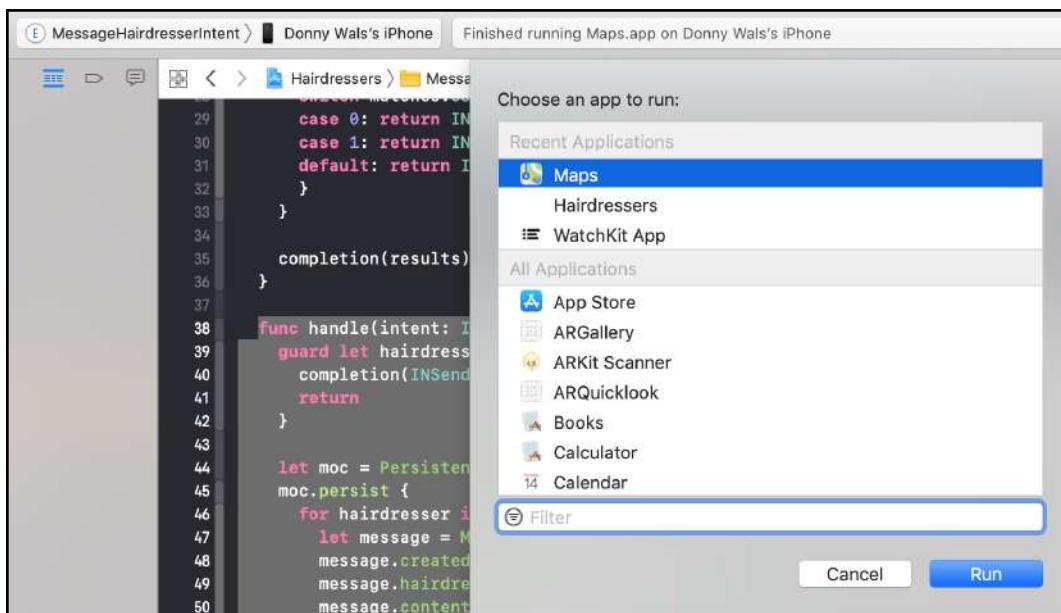
    let moc = PersistentHelper.shared.persistentContainer.viewContext
    moc.persist {
        for hairdresser in hairdressers {
            let message = Message(context: moc)
            message.createdAt = Date()
            message.hairdresser = hairdresser.displayName
            message.content = content
        }
    }

    completion(INSendMessageIntentResponse(code: .success,
        userActivity: nil))
}
```

```
    }  
}
```

All the preceding code does is create a new `Message` object for each recipient of the message and stores it in the Core Data database. If no recipients are found, or no message exists, the operation is considered failed. If everything works out well, the operation is completed successfully.

To test your extension, click on your app target in the top of the Xcode window and select your extension. When you attempt to run it, Xcode will ask you for an app to run it with, as shown in the following screenshot. You can pick **Hairdressers** if you want, but any app will do:



Before testing the extension, make sure that you have run **Hairdressers** at least once to make sure the app has permission to be integrated with Siri.

Even though it's pretty cool that you can now use Siri to send messages, the UI that Siri shows isn't that great. In the next section, you will learn how you can add a custom UI to your Siri intent.

Adding a custom UI to Siri

When your user interacts with Siri, they aren't always looking at their device. But when they are, it's desirable that the experience a user has when using your app through Siri looks and feels a lot like when they're directly interacting with your app. One of the tools to achieve this is using custom vocabularies. You can use a vocabulary to map user- and app-specific terms to Siri's vocabulary, as you have seen in the previous section.

Another way we can customize the Siri experience is through an **Intents UI Extension**. Whenever you add an **Intents Extension** to your project, Xcode asks you if you also want to add a corresponding UI extension. If you select this checkbox, you don't have to do anything to add the UI extension, since it's already there. However, if you didn't check the checkbox, you should add a new **Intents UI Extension** by clicking the + icon in the list of targets for your project, just as you did when you added the **Intents Extension** before.

A custom user interface for an intent works a lot like other UI extensions. When you create the extension, Xcode creates a storyboard, a view controller, and a .plist file. The .plist file is expected to specify all of the intents that this UI extension can handle. In the case of **Hairdressers**, this is just a single intent; `INSendMessageIntent`. Make sure to update the `Info.plist` for the UI extension accordingly.

If you intend to support multiple intents, it's often a good idea to split your extension up into various extensions, while grouping intents together based on how similar they are. This will make it easier to maintain your code in the long run, and it makes it easier to reason about the way your code and extensions work.

If you open the `IntentViewController` file, you'll find a method named `configureView(for:of:interactiveBehavior:context:completion:)`. This method is used to configure the UI extension's view, and you can remove it for now because you'll implement a slightly simpler version of this method later.

Since the UI extension is used for a messaging feature, it would be nice to show the Hairdressers that will end up receiving the message, and the message that they will receive. Add the following two `@IBOutlet` properties to the `IntentViewController`:

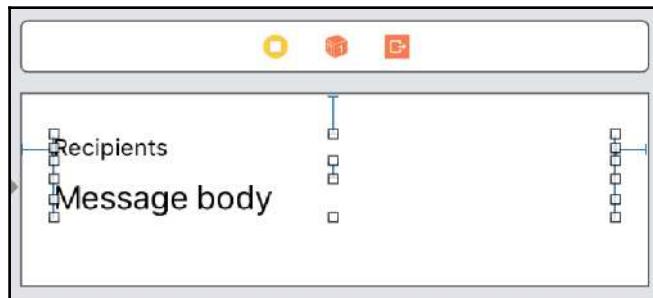
```
@IBOutlet var recipientsLabel: UILabel!
@IBOutlet var messageContentLabel: UILabel!
```

After adding the outlets, add the following method to the IntentViewController:

```
func configure(with interaction: INInteraction, context: INUIHostedViewContext, completion: ((CGSize) -> Void)) {  
  
    guard let messageIntent = interaction.intent as?  
    INSendMessageIntent,  
        let recipients = messageIntent.recipients  
    else { return }  
  
    recipientsLabel.text = recipients.map { $0.displayName  
}.joined(separator: ", ")  
    messageContentLabel.text = messageIntent.content  
  
    let viewWidth = extensionContext?.hostedViewMaximumAllowedSize.width  
    ?? 0  
    completion(CGSize(width: viewWidth, height: 100))  
}
```

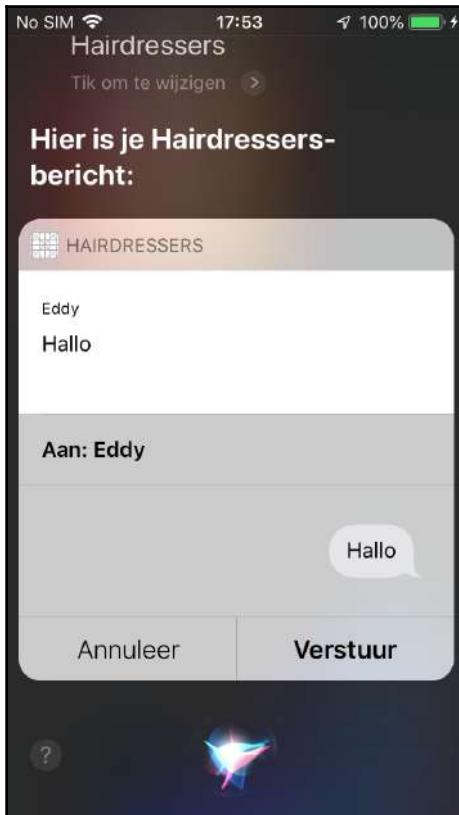
This implementation verifies that a message intent with recipients exists, and then populates the labels you just added accordingly. The code also determines at which width the UI should be displayed. Finally, the completion handler is called with the desired size at which Siri should display the custom UI. A value of 100 should be plenty of room to accommodate the message for now.

Next, open the storyboard file for the UI extension and add two labels. Lay them out as shown in the following screenshot. Don't forget to connect the outlets for these labels as well. You can set the text style for the message label to **Body**, and the recipients label can be set to **Caption** style:



After setting up the UI, run your UI extension instead of the intent extension, and try to send a message to one of the Hairdressers again.

You should see something that looks like the following UI:



The cool part is that the custom UI appears. There is one problem though. The message transcript is now shown twice: Once by the extension, and once through Siri itself. To prevent this from happening, all you need to do is make our view controller conform to `INUIHostedViewSiriProviding`. Add this protocol to the declaration of `IntentViewController`, and add the following property as well:

```
var displaysMessage = true
```

By doing this, Siri is now aware that you render your own version of the message transcript, so there is no need for Siri to also show the message. Now that you know how to implement a Siri intent and provide a custom UI for it, it's time to take your Siri integration one step further, and have a look at Siri Shortcuts.

Implementing Siri Shortcuts

If you have used iOS for a while, you may have noticed that in places such as the notification center, Siri suggests certain apps you might want to use at a certain time. Siri can do this because it continuously learns about the apps you use, when you use them, and where you are when you use them. By learning your behavior, Siri can make accurate predictions about what you might want to do next.

With the introduction of iOS 12, Apple has announced a new, powerful feature for Siri. This feature is called **Siri Shortcuts**, and it allows app developers to teach Siri about certain actions that a user performs in their apps. Siri then learns about these actions and offers them to the user at the appropriate time.

If you book a taxi home every day at a certain time, and the app you use has implemented Siri Shortcuts, Siri can offer you a shortcut to quickly book your taxi without having to go into the app. This is a great feature for users because it allows them to focus on the things that matter to them, without having to perform repetitive tasks inside of an app.

In this section, you will implement Siri Shortcuts for the Hairdressers app to enable users to quickly book appointments. There are two ways for apps to support shortcuts:

- Through `NSUserActivity` objects
- By donating `INInteraction` objects

First, you will implement the `NSUserActivity` method for adding shortcuts. Then you will define your own custom intent, so you can donate activities through `INInteraction` objects, and handle shortcuts in the background by implementing a second **Intents Extension**.

Implementing shortcuts through NSUserActivity

The simplest way to implement Siri Shortcuts is to use `NSUserActivity` objects. An `NSUserActivity` object contains information about the current activity that a user is performing, and can be used to achieve several goals, one of which is to implement Siri Shortcuts. To find out about other uses of `NSUserActivity`, have a look at Chapter 25, *Improved Discoverability with Spotlight and Universal Links*.

Your app must register the identifiers, in the `Info.plist` file, for the user activity objects that it will handle. Add a new entry to the `Info.plist` for `Hairdressers`, and name it **NSUserActivityTypes**. The object type for this key should be an array. In this array, you should add the identifier for the user activity object you are going to use for Siri shortcuts. Usually, this will be a reverse domain name kind of identifier, such as, for example, `com.donnywals.hairdressers.appointment`.

After adding the identifier to your `Info.plist`, you are ready to implement user activities in your app. To nicely encapsulate all logic related to the appointment booking activity, add an extension called `NSUserActivity` to the extensions folder, and implement the following extension in it:

```
extension NSUserActivity {
    static var identifierForAppointment: String {
        return "com.donnywals.hairdressers.appointment"
    }

    static func appointmentActivity() -> NSUserActivity {
        return NSUserActivity(activityType: identifierForAppointment)
    }
}
```

The preceding extension contains the identifier you entered earlier, and a convenience method to create a new user activity that uses the type identifier for an appointment booking activity.

Next, create a folder called `Helpers`, and add a file called `AppointmentShortcutHelper`. You will implement all the logic surrounding shortcuts in this file.

When a user books a new appointment, you should create a new user activity and pass it to Siri Shortcuts. Passing shortcuts to Siri Shortcuts is also called donating a shortcut. Every user activity you donate to Siri needs to have a `title` property, a `userInfo`, a `persistentIdentifier`, and it needs to be marked as `isEligibleForPrediction`.

The following helper method creates a user activity object and should be added to the `AppointmentShortcutHelper`:

```
static func activityForAppointment(_ appointment: Appointment) ->
    NSUserActivity {
    let userActivity = NSUserActivity.appointmentActivity()
    let title = "Book an appointment with \u2028\ufe0f(\u2028\ufe0fappointment.hairdresser!)"
    for \u2028\ufe0f(appointment.day!)"

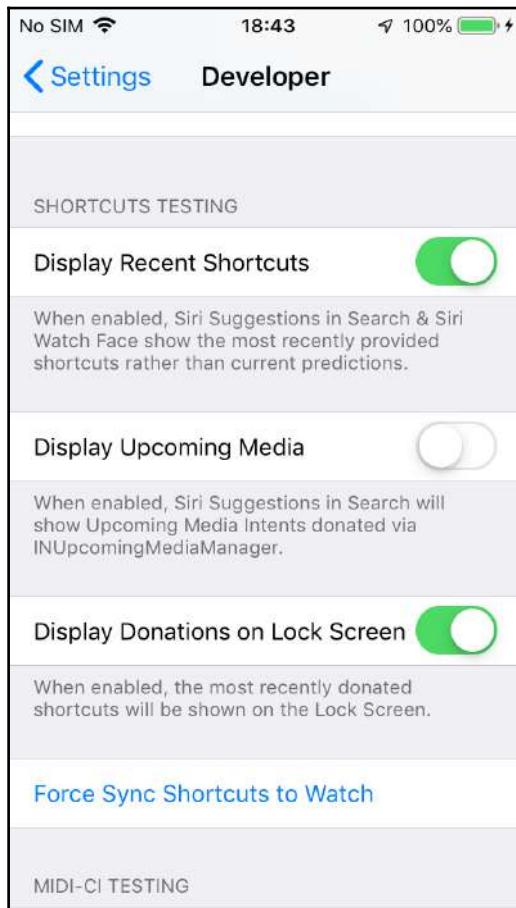
        userActivity.requiredUserInfoKeys = ["hairdresser", "day"]
        userActivity.userInfo = ["hairdresser": appointment.hairdresser!,
        "day": appointment.day!]
        userActivity.title = title
        userActivity.isEligibleForPrediction = true
        userActivity.persistentIdentifier = "\u2028\ufe0f(appointment.hairdresser!)-
        \u2028\ufe0f(appointment.day!)"
        userActivity.suggestedInvocationPhrase = title

    return userActivity
}
```

The preceding block of code creates a user activity. Note that the sample code uses some force unwrapping with a `!` symbol used to keep the code brief and simple. Make sure to assess whether this is appropriate in your own app, and preferably use safe unwrapping instead. To use and donate the user activity that the preceding snippet creates, add the following code to `AddAppointmentViewController` in the `didTapSave()` method, right before `dismiss(animated:completion:)` is called:

```
let activity =
AppointmentShortcutHelper.activityForAppointment(appointment)
self.userActivity = activity
```

To make an activity the current activity, you need to assign it as the current user activity for the current view controller. If you run the app now, creating a new appointment will donate that appointment to Siri. Normally, Siri will begin showing donations once the user has performed a certain action many times, so Siri is quite sure that it makes sense for your user to see the action offered to them. For debugging purposes, you can go to **Settings** | **Developer**, scroll down a bit, and then enable **Display Recent Shortcuts** and **Display Donations on Lock Screen**, as shown in the following screenshot:



Doing this will immediately show your new donated shortcuts in the notification center and on the lock screen, as shown in the following screenshot:



If you only donate your shortcuts to Siri, you're not quite done yet. When a user taps on your shortcut, your application should receive the user activity object and then perform the action the user wants to perform—in this case, booking a repeat appointment with a certain hairdresser.

Add the following helper method to the `AppointmentShortcutHelper` to convert a user activity to an appointment and store it in Core Data:

```
static func storeAppointmentForActivity(_ userActivity:  
NSUserActivity) {  
    guard let userInfo = userActivity.userInfo,  
          let hairdresser = userInfo["hairdresser"] as? String,  
          let day = userInfo["day"] as? String  
    else { return }  
}
```

```
let moc = PersistentHelper.shared.persistentContainer.viewContext

moc.persist {
    let appointment = Appointment(context: moc)
    appointment.day = day
    appointment.hairdresser = hairdresser
}
}
```

When your app is asked to handle a user activity, the `application(_:continue:restorationHandler:)` method is called on your `AppDelegate`. Add the following implementation for this method to check whether the received user activity is related to booking an appointment, and pass it on to the helper method to store a new appointment:

```
func application(_ application: UIApplication, continue userActivity: NSUserActivity, restorationHandler: @escaping ([UIUserActivityRestoring]?) -> Void) -> Bool {

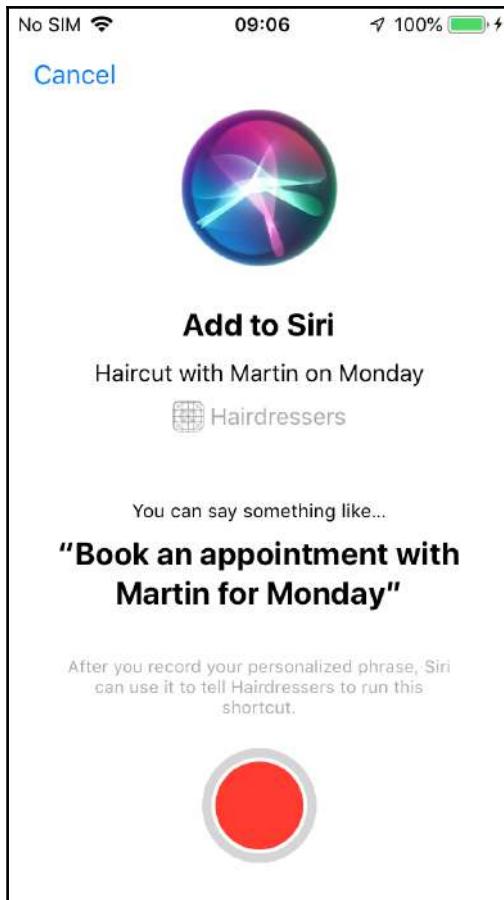
    guard userActivity.userActivityType ==
        NSUserActivity.identifierForAppointment
        else { return false }

    AppointmentShortcutHelper.storeAppointmentForActivity(userActivity)

    return true
}
```

After you implement this code, go ahead and book an appointment in the Hairdressers app. You should see your appointment appear in the list of suggested shortcuts, and when you tap on your shortcut, a new appointment should immediately be created.

Users can also add their own voice command for the shortcuts you donate. Users can do this by navigating to the following page in the settings app: **Hairdressers | Siri & Search | My "Hairdressers" Shortcuts**. A user can then tap on a shortcut to record their own phrase, which can then be used to execute the shortcut they chose, as shown in the following screenshot:



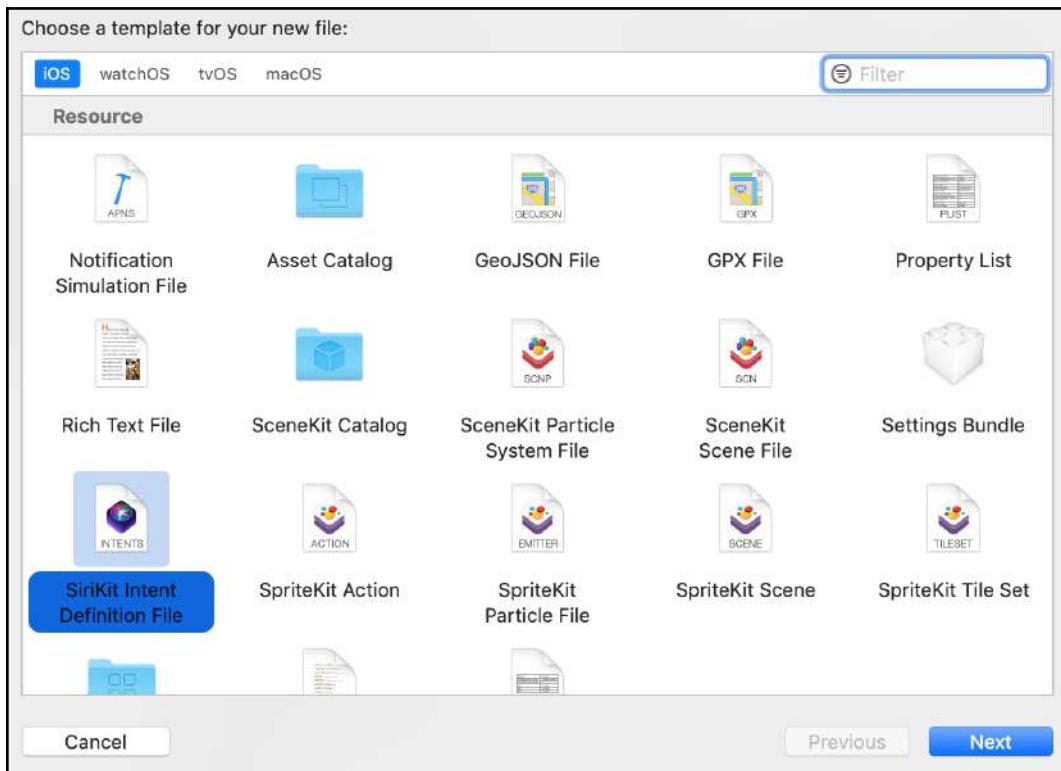
When you donate shortcuts using a user activity, your app will be launched to handle the shortcut. If you want to handle shortcuts in the background, you can create a custom intent and **Intents extension** to achieve this. Let's see how.

Donating shortcuts with INteractions

Donating shortcuts with user activities is pretty straightforward and powerful. The only downside is that your app has to be launched to handle shortcuts when the user executes them. Sometimes this might be the behavior you need, but for the case of booking an ap

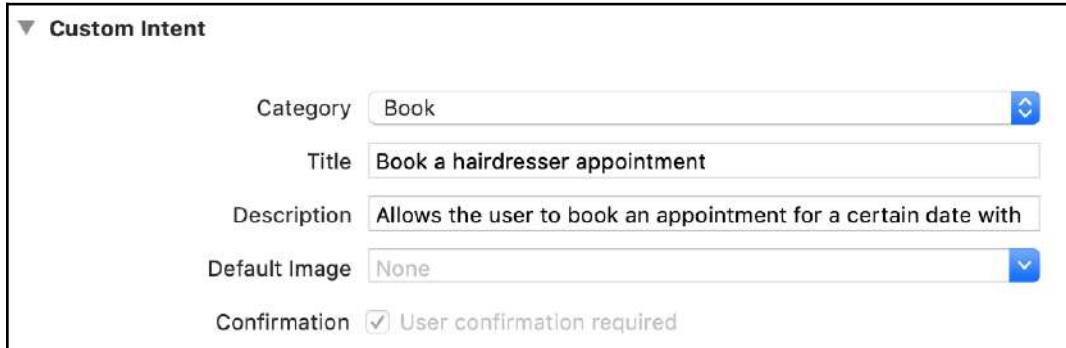
To achieve this, you need to implement a custom intent, and an extension that can handle it. Let's set up the custom intent and prepare the app for donating and potentially handling the intent. After doing this, you will see how to implement an **Intents extension** that handles your custom intent.

The first step to implement your own custom intent is to create an intent definition file. Add a new file to your project and choose the **SiriKit Intent Definition File** type from the list of available files, as shown in the following screenshot:



When you select this file in Xcode, the intent editor is opened. In this editor, you can create your custom intents and configure them. Click the + icon in the bottom left corner of the editor to create a new intent. Name your new intent BookAppointment.

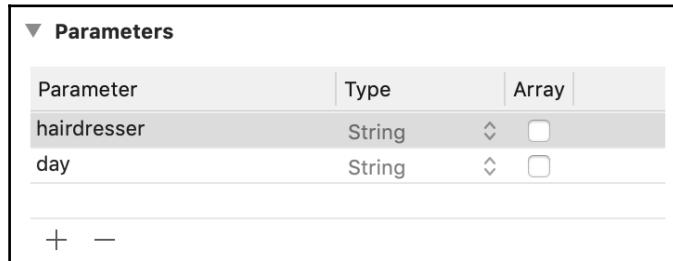
After creating your intent, you need to choose a category for your intent. There are many available categories to choose from. You should always pick the category that fits your needs as closely as possible. In this case, the **Book** category is a perfect choice. The next step is to give your intent a title and a description, as shown in the following screenshot:



The parameters for your intent are configured in the **Parameters** section. The appointment booking intent should have two parameters:

- hairdresser (String)
- day (String)

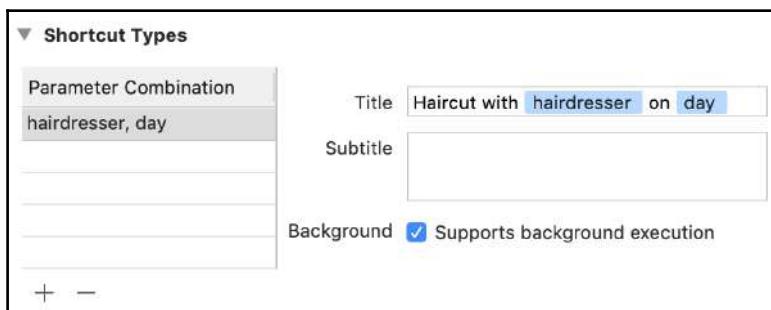
These parameters should appear in Xcode as follows:



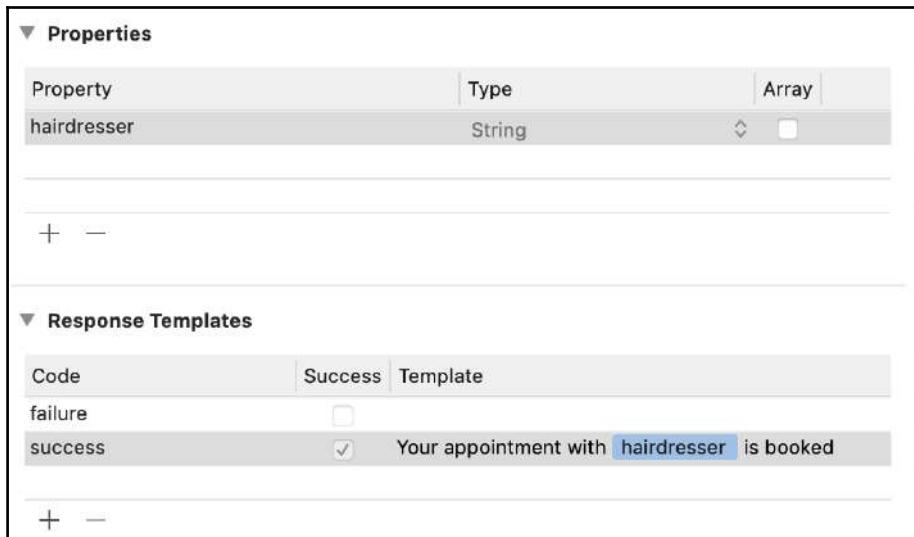
Xcode will generate several subclasses for your custom intent after you have configured it. The parameters you specify in the **Parameters** section will be available as properties on your custom intent.

Now that you have the parameters prepared, you need to configure the shortcut that you want to donate. Every shortcut you base on the intent you just created will have a unique combination of parameters. These parameters can be used to generate a dynamic title for your shortcut.

Add a parameter combination for the hairdresser and day parameters, and create a title that uses these parameters. You can start typing the message, and Xcode will automatically suggest the parameters for you. The final shortcut should look like the following screenshot:



After configuring the intent and shortcut, you can set up response parameters. You can use these parameters to create a helpful response that confirms information for the user. In this case, you can add a `hairdresser` property and use that to verify a user's appointment, as shown in the following screenshot:



Now that the custom intent is set up, you can update the `AppointmentShortcutHelper` to donate an `INInteraction`, which contains the custom intent, to Siri. Add the following implementation for this helper to `AppointmentShortcutHelper`:

```
static func donateInteractionForAppointment(_ appointment:  
Appointment) {  
    let intent = BookAppointmentIntent()  
    intent.hairdresser = appointment.hairdresser  
    intent.day = appointment.day  
    let title = "Book an appointment with \(appointment.hairdresser!)  
for \(appointment.day!)"  
    intent.suggestedInvocationPhrase = title  
  
    let interaction = INInteraction(intent: intent, response: nil)  
    interaction.donate(completion: nil)  
}
```

Xcode has generated a `BookAppointmentIntent` class based on your custom intent definition. As mentioned before, this intent has the properties that you have defined in the editor, so that you can properly configure your intent.

After configuring the intent, it is wrapped in an `INInteraction` object. To donate the shortcut to Siri, you must call `donate(completion:)` on the interaction object. Next, replace the user activity-based donation code in `AddAppointmentViewController` with the following line:

```
AppointmentShortcutHelper.donateInteractionForAppointment(appointment)
```

If your application does happen to be launched by Siri to handle the intent, you should do so in the `application(_:continue:restorationHandler:)` method in `AppDelegate`. The user activity type for your custom intent will be the name of the intent you created. So, if you created an intent called `BookAppointment`, that will be the user activity type that Siri will use. Update the implementation of `application(_:continue:restorationHandler:)` as follows:

```
func application(_ application: UIApplication, continue userActivity:  
NSUserActivity, restorationHandler: @escaping  
([UIUserActivityRestoring]?) -> Void) -> Bool {  
  
    var allowedActivities = [NSUserActivity.identifierForAppointment,  
"BookAppointment"]  
  
    guard allowedActivities.contains(userActivity.activityType)  
    else { return false }
```

```
    AppointmentShortcutHelper.storeAppointmentForActivity(userActivity)
    return true
}
```

This wraps up the configuration of your app. The next step is to create an **Intents Extension** that can handle the new custom intent. Since you have already done this before, the processes won't be explained step by step. Instead, make sure you do all of the following:

1. Create an **Intents Extension**.
2. Enable **App Groups** for the extension.
3. Add the intent definition, managed object context extension, data helpers, and Core Data model to the new extension's target.
4. Configure the extension to support the `BookAppointment` intent.
5. Remove the boilerplate code from the `IntentHandler` class.

Since the `IntentHandler` for the custom intent is only meant to handle a single intent, add the following implementation for the `IntentHandler` class:

```
class IntentHandler: INExtension, BookAppointmentIntentHandling {

    override func handler(for intent: INIntent) -> Any {
        return self
    }

    func handle(intent: BookAppointmentIntent, completion: @escaping
    (BookAppointmentIntentResponse) -> Void) {
        guard let hairdresser = intent.hairdresser, let day = intent.day
        else {
            completion(BookAppointmentIntentResponse(code: .failure,
            userActivity: nil))
            return
        }

        let moc = PersistentHelper.shared.persistentContainer.viewContext

        moc.persist {
            let appointment = Appointment(context: moc)
            appointment.day = day
            appointment.hairdresser = hairdresser

            let response = BookAppointmentIntentResponse(code: .success,
            userActivity: nil)
            response.hairdresser = hairdresser
            completion(response)
        }
    }
}
```

```
    }  
}
```

If you book a new appointment in the Hairdressers app and invoke the Siri Shortcut later, the **Intents extension** will be used to handle your intent rather than having to open the app. This is great, because it allows the user to go on with what they were doing, without having to look at your app for confirmation.

Now that your extension is in place, go ahead and play with your Siri Shortcuts a bit. The work for this chapter is completed—you have created an application that is tightly integrated with Siri in order to streamline the user's experience with your app.

Summary

This chapter has taught you a lot about how you can integrate your app with Siri. First, you learned how to handle certain predefined intents, such as sending messages by adding an **Intents Extension**. Then you implemented a custom UI for your extension to enhance the Siri experience.

Next, you learned how to enhance the user's experience by donating Siri Shortcuts. Shortcuts integrate deeply with iOS, which means that Siri picks up and shows the shortcuts added by you at the most appropriate time for the user. You also learned that users can record their own phrases for shortcuts, so they can trigger a shortcut by simply asking Siri.

The next chapter will teach you how you can integrate media like video, audio, and the camera in your app. So make sure to skip on to the next chapter when you're done playing with your Siri integration.

Questions

1. How does Siri understand a user's request?
 - a) It has templates for the user's requests.
 - b) Siri uses natural language processing.
 - c) Siri uses the current time to determine what a user means.

2. How are app extensions launched?
 - a) App extensions are launched by the app that owns the extension.
 - b) App extensions are launched by the currently active app.
 - c) App extensions are launched by the extensions framework.
3. How can you share data between an app and an extension?
 - a) By adding the required files to both targets.
 - b) By implementing app groups.
 - c) Both of the above.
4. What are the three stages of handling a Siri intent?
 - a) Resolve, Confirm, Handle.
 - b) Confirm, Resolve, Handle.
 - c) Handle, Resolve, Confirm.
5. How do you make sure Siri does not render its default UI?
 - a) By providing your own UI.
 - b) By conforming the `IntentViewController` to the `INUIHostedViewSiriProviding` protocol, and setting `displaysMessage` to `true`.
 - c) By setting `showsDefaultUI` to `false` in the UI extension's `Info.plist`.
6. How can you donate actions to Siri?
 - a) Through user activities and interaction objects
 - b) With the Siri Shortcuts framework
 - c) By adding them to a Core Data database
7. How can a user add their own voice commands for Siri Shortcuts?
 - a) Through the settings app.
 - b) The app must provide its own audio recording.
 - c) Siri automatically asks this when an intent is donated.

Further reading

- The Intents documentation is available at <https://developer.apple.com/documentation/sirikit/inextension>
- The SiriKit documentation is available at <https://developer.apple.com/documentation/sirikit>

21

Using Media in Your App

lot of the apps that people use every day make use of media in some way. Some apps show photos and videos in a user's feed. Other apps focus on playing audio or video, while there are also apps that allow users to record media and share it with their peers. You can probably name at least a couple of very well known apps that make use of such media in one way or the other.

Because media has such a significant presence in people's daily life, it's good to know how you can integrate media into your own apps. iOS has excellent support for media playback and offers several different ways to create and consume different types of media. Some ways provide less flexibility, but are more straightforward to implement. Others are more complex, but provide significant power to you as a developer.

In this chapter, you will learn about several ways to play and record media on iOS. You will learn how to play and record video, play audio, take pictures, and you'll even learn how to apply filters to images with Apple's Core Image framework. This chapter covers the following topics:

- Playing audio and video
- Recording video and taking pictures
- Manipulating photos with Core Image

By the end of this chapter, you will have a great foundation that you can build on to create engaging experiences for your users, allowing them to not only view content, but also to create their own content in your app.

Playing audio and video

To make playing audio and video files as simple and straightforward as can be, Apple has created the `AVFoundation` framework. This framework contains a lot of helper classes that provide very low-level control over how iOS plays audio and video files. You can use `AVFoundation` to build a rich, custom media player with as many features as you need for your purpose.

If you're looking for a simpler way to integrate media into your app, the `AVKit` framework might be what you need. `AVKit` contains several helpers that build upon the `AVFoundation` components to provide an excellent default player that supports many features like subtitles, Airplay, and more.

In this section, you will learn how to implement a simple video player with the `AVPlayerViewController` from the `AVKit` framework. You will also implement a more complex audio player with `AVFoundation` components that plays audio in the background and displays, on the lock screen, the audio track currently being played.

To follow along with the samples, you should open the `MediaPlayback_start` project in this chapter's code bundle. The starter app contains a straightforward interface with a tab bar and two pages. You will implement a video player on one page, and the audio player on the other page. The audio page comes with some predefined controls and actions that you will implement later.

Creating a simple video player

The first thing you need to do to implement a video player is to obtain a video file. You can use any video that is encoded in the `h.264` format. A good sample video is the **Big Buck Bunny** sample movie that was created by the Blender Foundation. You can find this video at the following URL: <http://bbb3d.renderfarming.net/download.html>. If you want to use this video to practice with, make sure to download the 2D version of the video.

As stated before, you will implement the video player using the `AVPlayerViewController`. This view controller provides a convenient wrapper around several components from `AVFoundation`, and also provides default video controls, so you don't have to build your entire video player from scratch, as you will do for the audio player later.

The `AVPlayerViewController` is highly configurable, which means that you can choose whether the player supports Airplay, shows playback controls, whether it should be full screen when a video plays, and more. For a complete list of configurable options, you can refer to Apple's `AVPlayerViewController` documentation.

Once you have found your test video, you should add it to the **MediaPlayback** project and ensure that the video is added to the app target. After doing this, open `VideoViewController.swift` and add the following line to import `AVKit`:

```
import AVKit
```

You should also add a property to `VideoViewController` to hold on to your video player instance. Add the following line to the `VideoViewController` class to do this:

```
let playerController = AVPlayerViewController()
```

Since the `AVPlayerViewController` is a `UIViewController` subclass, you should add it to the `VideoViewController` as a child view controller. Doing this will make sure that `VideoViewController` forwards any view controller lifecycle events, such as `viewDidLoad()`, along with any changes in trait collections and more to the video player. To do this, add the following code to the `viewDidLoad()` method in `VideoViewController`:

```
// 1
addChild(playerController)
playerController.didMove(toParent: self)

// 2
view.addSubview(playerController.view)

let playerView = playerController.view!
playerView.translatesAutoresizingMaskIntoConstraints = false

NSLayoutConstraint.activate([
    playerView.widthAnchor.constraint(equalTo: view.widthAnchor,
    constant: -20),
    playerView.heightAnchor.constraint(equalTo: playerView.widthAnchor,
    multiplier: 9/16),
    playerView.centerXAnchor.constraint(equalTo: view.centerXAnchor),
    playerView.centerYAnchor.constraint(equalTo: view.centerYAnchor)])
```

The previous code snippet adds the video player to the video view controller as a child view controller. When you add a view controller as a child view controller, you must always call `didMove(toParent:)` on the child controller to make sure it knows that it has been added as a child view controller to another view controller. After adding the video player as a child view controller, the video player's view is added as a subview for the video view controller, and some constraints are set up to position the player view.

This is all you need to do to create an instance of the video player and make it appear in your view controller. The last step is to obtain a reference to your video file, create an `AVPlayer` that has a reference to the video file, and assign it to the player. Add the following code to do this:

```
let url = Bundle.main.url(forResource: "samplevideo", withExtension:  
    "mp4")!  
playerController.player = AVPlayer(url: url)
```

The preceding code looks for a video file called `samplevideo.mp4` and obtains a URL for that file. It then creates an instance of `AVPlayer` that points to that video file, and assigns it to the video player. The `AVPlayer` object is responsible for actually playing the video file. The `AVPlayerViewController` instance uses the `AVPlayer` instance to play the video, and manages the actual playback of the video internally.

If you run your app after adding the player this way, you will find that the video plays perfectly well, and that you have access to all the controls you might need. This is a great demonstration of how simple it is to add basic media integration to your app. The next step is a little more complex. You will directly use an `AVAudioPlayer` instance to play an audio file that is controlled through several custom media controls. The player will even play audio in the background and integrate with the lock screen to show information about the current file. In other words, you will build a simple audio player that does everything a user would expect it to do.

Creating an audio player

Before you can implement your audio player, you will need to obtain some `.mp3` files you wish to use in your player. If you don't have any audio files on your computer, you can get some files from The Free Music Archive website, available at <https://freemusicarchive.org/about>, to obtain a couple of free songs that you would like to use for playback. Make sure to add them to the **MediaPlayer** Xcode project and ensure that they are included in the app target.

You will build the audio player using the following steps:

1. Implementing the necessary controls to start and stop the player, and navigate to the next and previous songs.
2. Implementing the time scrubber.
3. Reading the file's metadata and show it to the user.

The user interface, outlets, and actions are already set up, so make sure to familiarize yourself with the existing code before following along with the implementation of the audio player.

Implementing basic audio controls

Before you implement the audio player code, you will need to do a little bit of housekeeping. To be able to play audio, you need a list of the files that the player will play. In addition to this list, you also need to keep track of what song the user is currently playing, so you can determine the next and previous songs. Lastly, you also need to have the audio player itself. Instead of using a pre-built component, you will build your own audio player using an `AVAudioPlayer` object. The `AVAudioPlayer` is perfect for implementing a simple audio player that plays a couple of local `.mp3` files. It offers some convenient helper methods to easily adjust the player's volume, seek to a specific timestamp in the song, and more.

Define the following properties in `AudioViewController.swift`:

```
let files = ["one", "two", "three"]
var currentTrack = 0
var audioPlayer: AVAudioPlayer!
```

Make sure to replace the `files` array with the filenames that you use for your own audio files. The `audioPlayer` does not have a value yet at this point. You will set up the audio player next.

Before you can play audio, you need to obtain a reference to a media file and provide this reference to an `AVAudioPlayer` object. Any time you want to load a new media file, you will have to create a new instance of the audio player, since you can't change the current file once a file is playing. Add the following helper method to the `AudioViewController` to load the current track and create an `AVAudioPlayer` instance:

```
func loadTrack() {
    let url = Bundle.main.url(forResource: files[currentTrack],
    withExtension: "mp3")!
```

```
    audioPlayer = try! AVAudioPlayer(contentsOf: url)
    audioPlayer.delegate = self
}
```

This method reads the file name for the current track and retrieves the local URL for it. This URL is then used to create and set the `audioPlayer` property on the `AudioViewController`. The view controller is also assigned as the delegate for the audio player. You won't implement any of the delegate methods just yet, but you can add the following extension to make `AudioViewController` conform to the `AVAudioPlayerDelegate` protocol to ensure your code compiles:

```
extension AudioViewController: AVAudioPlayerDelegate {  
}
```

You will implement one of the `AVAudioPlayerDelegate` methods when you add support for navigating to the next and previous tracks.

Add the following two methods to the audio view controller to add support for playing and pausing the current audio file:

```
func startPlayback() {
    audioPlayer.play()
    playPause.setTitle("Pause", for: .normal)
}  
  
func pausePlayback() {
    audioPlayer.pause()
    playPause.setTitle("Play", for: .normal)
}
```

These methods are relatively straightforward. They call the audio player's `play()` and `pause()` methods and update the button's label, so it reflects the current player state. Add the following implementation for `playPauseTapped()` so the play and pause methods get called when the user taps the play/pause button:

```
@IBAction func playPauseTapped() {
    if audioPlayer.isPlaying {
        pausePlayback()
    } else {
        startPlayback()
    }
}
```

If you run the app now, you can tap the play/pause button to start and stop the currently playing file. Make sure your device is not in silent mode, because the audio for your app is muted when the device is in silent mode. You will learn how to fix this when you implement the ability to play audio in the background. The next step is to add support for playing the next and previous tracks. Add the following two implementations to `AudioViewController` to do this:

```
@IBAction func nextTapped() {
    currentTrack += 1
    if currentTrack >= files.count {
        currentTrack = 0
    }
    loadTrack()
    audioPlayer.play()
}

@IBAction func previousTapped() {
    currentTrack -= 1
    if currentTrack < 0 {
        currentTrack = files.count - 1
    }
    loadTrack()
    audioPlayer.play()
}
```

The preceding code adjusts the current track index, loads the new track, and immediately plays it. Note that every time the user taps on the next or previous button, a fresh audio player has to be created by calling `loadTrack()`. If you run the app now, you can play audio, pause it, and skip to the next or previous tracks.

When you allow a full song to play, it won't yet advance to the next song afterward. To implement this, you need to add an implementation for the `audioPlayerDidFinishPlaying(_:successfully:)` method from `AVAudioPlayerDelegate`. Add the following implementation to call `nextTapped()`, so the next song automatically plays when the current song finishes:

```
func audioPlayerDidFinishPlaying(_ player: AVAudioPlayer, successfully
flag: Bool) {
    nextTapped()
}
```

Now that the first features are implemented, the next step is to implement the time scrubber that shows the current song's progress and allows the user to adjust the playhead's position.

Implementing the time scrubber

The user interface for the audio player app already contains a scrubber that is hooked up to the following three actions in the view controller:

- `sliderDragStart()`
- `sliderDragEnd()`
- `sliderChanged()`

When an audio file is playing, the scrubber should automatically update to reflect the current position in the song. However, when a user starts dragging the scrubber, it should not update its position until the user has chosen the scrubber's new position. When the user is done dragging the scrubber, it should adjust itself based on the song's progress again. Any time the value for the slider changes, the audio player should adjust the playhead, so the song's progress matches that of the scrubber.

Unfortunately, the `AVAudioPlayer` object does not expose any delegate method to observe the progress of the current audio file. To update the scrubber regularly, you can implement a timer that updates the scrubber to the audio player's current position every second. Add the following property to the `AudioViewController`, so you can hold on to the timer after you have created it:

```
var timer: Timer?
```

Also, add the following two methods to `AudioViewController` as a convenient way to start the timer when the user starts dragging the scrubber, or when a file starts playing, and stop it when a user starts dragging the scrubber or to preserve resources when the playback is paused:

```
func startTimer() {
    timer = Timer.scheduledTimer(withTimeInterval: 1, repeats: true) {
        [unowned self] timer in
        self.slider.value = Float(self.audioPlayer.currentTime /
            self.audioPlayer.duration)
    }
}

func stopTimer() {
    timer?.invalidate()
}
```

Add a call to `startTimer()` in the `startPlayback()` method and a call to `stopTimer()` in the `stopPlayback()` method. If you run the app after doing this, the scrubber will immediately begin updating its position when a song starts playing. However, scrubbing does not work yet. Add the following implementations for the scrubber actions to enable manual scrubbing:

```
@IBAction func sliderDragStart() {
    stopTimer()
}

@IBAction func sliderDragEnd() {
    startTimer()
}

@IBAction func sliderChanged() {
    audioPlayer.currentTime = Double(slider.value) *
    audioPlayer.duration
}
```

The preceding methods are relatively simple, but they provide a very powerful feature that immediately makes your homemade audio player feel like an audio player you might use every day. The final step for implementing the audio player's functionality is to display metadata about the current song.

Displaying song metadata

Most `.mp3` files contain metadata in the form of `ID3` tags. These metadata tags are used by applications such as iTunes to extract information about a song and display it to the user, as well as to categorize a music library or filter it. You can gain access to an audio file's metadata through code, by loading the audio file into an `AVPlayerItem` object and extracting the metadata for its internal `AVAsset` instance. An `AVAsset` object contains information about a media item, such as its type, location, and more. When you load a file using an `AVPlayerItem` object, it will automatically create a corresponding `AVAsset` object for you.

A single asset can contain loads of metadata in the metadata dictionary. Luckily, Apple has captured all of the valid ID3 metadata tags in the `AVMetadataIdentifier` object, so once you have extracted the metadata for an `AVAsset`, you can loop over all of its metadata to filter out the data you need. The following method does this, and sets the extracted values on the `titleLabel` variable of `AudioViewController`, as shown here:

```
func showMetadataForURL(_ url: URL) {
    let mediaItem = AVPlayerItem(url: url)
    let metadata = mediaItem.asset.metadata
    var information = [String]()

    for item in metadata {
        guard let identifier = item.identifier
        else { continue }

        switch identifier {
        case .id3MetadataTitleDescription, .id3MetadataBand:
            information.append(item.value?.description ?? "")
        default:
            break
        }
    }

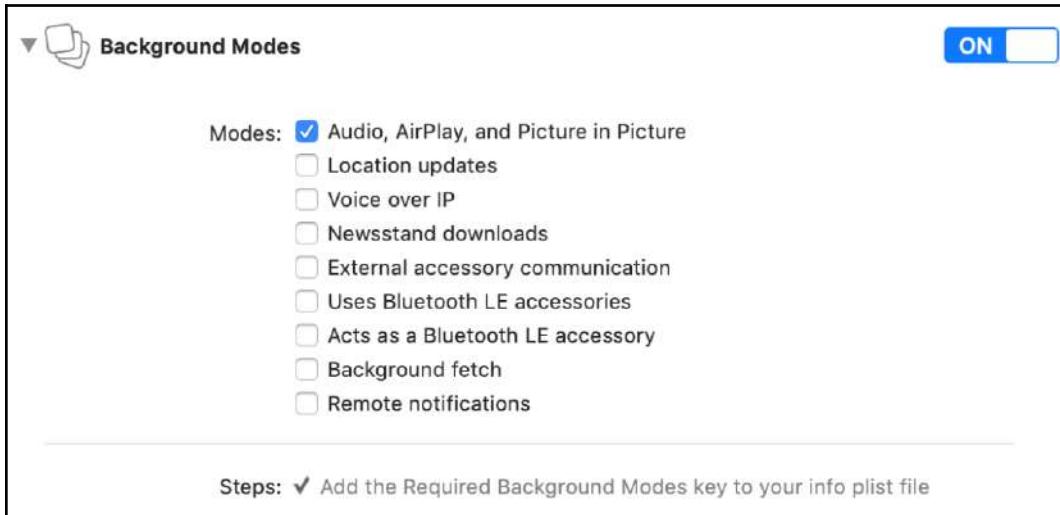
    let trackTitle = information.joined(separator: " - ")
    titleLabel.text = trackTitle
}
```

Make sure to add a call to this method from `loadTrack()`, and pass the audio file's URL that you obtain in `loadTrack()` to `showMetadataForURL(_:_)`. If you run your app now, your basic functionality should be all there. The metadata should be shown correctly, the scrubber should work, and you should be able to skip songs or pause the playback.

Even though your media player seems to be pretty much done at this point, did you notice that the music pauses when you send the app to the background? To make your app feel more like a real audio player, you should implement background audio playback and make sure the currently playing song is presented on the user's lock screen, similar to how the native music app for iOS works. This is precisely the functionality you will add next.

Playing media in the background

On iOS, playing audio in the background requires special permissions that you can enable in your app's **Capabilities** tab. If you enable the **Background modes** capability, you can select the **Audio**, **Airplay**, and **Picture in Picture** option to make your app eligible for playing audio in the background. The following screenshot shows the enabled capability for playing audio in the background:



If you want to add proper support for background audio playback, there are three features you need to implement:

- Set up an audio session, so audio continues playing in the background.
- Submit metadata to the "now playing" info center.
- Respond to playback actions from remote sources, such as the lock screen.

You can set up the audio session for your app with just two lines of code. When you create an audio session, iOS will treat the audio played by your app slightly differently. For instance, your songs will play even if the device is set to silent. It also makes sure your audio is played when your app is in the background, if you have the proper capabilities set up. Add the following code to `viewDidLoad()` to set up an audio session for the app:

```
try? AVAudioSession.sharedInstance().setCategory(.playback, mode:  
.default, options: [.allowAirPlay])  
try? AVAudioSession.sharedInstance().setActive(true, options: [])
```

The second feature to add is to supply information about the currently playing track. All information about the currently playing media file should be passed to the `MPNowPlayingInfoCenter` object. This object is part of the `MediaPlayer` framework, and is responsible for showing the user information about the currently playing media file on the lock screen and in the command center. Before you pass information to the "now playing" info center, make sure to import the `MediaPlayer` framework at the top of the `AudioViewController.swift` file.

Next, add the following line of code to `viewDidLoad()`:

```
NotificationCenter.default.addObserver(self, selector:  
    #selector(updateNowPlaying), name:  
    UIApplication.didEnterBackgroundNotification, object: nil)
```

In the documentation for `MPNowPlayingInfoCenter`, Apple states that you should always pass the most recent now playing information to the info center when the app goes to the background. To do this, the audio view controller should listen to the `UIApplication.didEnterBackgroundNotification` notification, so it can respond to the app going to the background. Add the following implementation for the `updateNowPlaying()` method to the `AudioVideoController`:

```
@objc func updateNowPlaying() {  
    var nowPlayingInfo = [String: Any]()  
    nowPlayingInfo[MPMediaItemPropertyTitle] = titleLabel.text ??  
    "untitled"  
    nowPlayingInfo[MPNowPlayingInfoPropertyElapsedPlaybackTime] =  
    audioPlayer.currentTime  
    nowPlayingInfo[MPMediaItemPropertyPlaybackDuration] =  
    audioPlayer.duration  
  
    MPNowPlayingInfoCenter.default().nowPlayingInfo = nowPlayingInfo  
}
```

The preceding code configures a dictionary with metadata about the currently playing file, and passes it to the "now playing" info center. This method is called automatically when the app goes to the background, but you should also update the "now playing" information when a new song begins playing. Add a call to `updateNowPlaying()` in the `loadTrack()` method to make sure the "now playing" information is updated whenever a new track is loaded.

The next and final step is to respond to remote commands. When the user taps the play/pause button, next button, or previous button on the lock screen, this is sent to your app as a remote command. You should explicitly define the handlers that should be called by iOS when a remote command occurs. Add the following method to `AudioViewController` to add support for remote commands:

```
func configureRemoteCommands() {
    let commandCenter = MPRemoteCommandCenter.shared()

    commandCenter.playCommand.addTarget { [unowned self] event in
        guard self.audioPlayer.isPlaying == false
        else { return .commandFailed }

        self.startPlayback()
        return .success
    }

    commandCenter.pauseCommand.addTarget { [unowned self] event in
        guard self.audioPlayer.isPlaying
        else { return .commandFailed }

        self.pausePlayback()
        return .success
    }

    commandCenter.nextTrackCommand.addTarget { [unowned self] event in
        self.nextTapped()
        return .success
    }

    commandCenter.previousTrackCommand.addTarget { [unowned self] event in
        self.previousTapped()
        return .success
    }

    UIApplication.shared.beginReceivingRemoteControlEvents()
}
```

The preceding code obtains a reference to the remote command center and registers several handlers. It also calls `beginReceivingRemoteControlEvents()` on the application object to make sure it receives remote commands. Add a call to `configureRemoteCommands()` in `viewDidLoad()`, to make sure that the app begins receiving remote commands as soon as the audio player is configured.

Try to run your app and send it to the background. You should be able to control media playback from both the control center and the lock screen. The visible metadata should correctly update when you skip to the next or previous song, and the scrubber should accurately represent the current position of playback in the song.

At this point, you have implemented a reasonably complete audio player that has pretty sophisticated behaviors. The next step in your exploration of media on iOS is to discover how you can take pictures and record video.

Recording video and taking pictures

In addition to playing existing media, you can also make apps that allow users to create their own content. In this section, you will learn how you can use a built-in component to enable users to take a picture. You will also learn how you can use a raw video feed to record a video. If you want to follow along with the samples in this section, make sure to grab the starter project for **Captured** from this chapter's code bundle.

The starter project contains a couple of view controllers and some connected outlets and actions. Note that there is a `UIViewController` extension in the project, too. This extension includes a helper method that makes displaying an alert to the user a little bit simpler. This extension will be used to show an alert that informs the user when their photo or video is stored to the camera roll.

Since a user's camera and photo library are considered very privacy-sensitive, you need to make sure that you add the following privacy-related keys to the app's `Info.plist`:

- **Privacy - Camera Usage Description:** this property is required in order to access the camera so you can take pictures and record video.
- **Privacy - Microphone Usage Description:** you must add this property so that your videos record audio, as well as images.
- **Privacy - Photo Library Additions Usage Description:** this property allows you to write photos to the user's photo library.

Make sure to provide a good description for the privacy keys, so the user knows why you need access to their camera, microphone, and photo library. The better your description is, the more likely the user is to allow your app to access the associated privacy-sensitive information. After adding the keys, you are ready to see how you can take a picture using the built-in `UIImagePickerController` component of the `UIKit`.

Taking and storing a picture

When you need a user to supply an image, they can do this by either selecting an image from their photo library, or by taking a picture with the camera. The `UIImagePickerController` supports both ways of picking an image. In this sample, you will learn how you can allow users to take an image using the camera. Changing the example to allow users to select an image from their photo library should be trivial, as long as you remember to add the **Privacy - Photo Library Usage Description** key to your `Info.plist`.

Add the following implementation for `viewDidLoad()` to the `ImageViewController` class:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let imagePicker = UIImagePickerController()
    imagePicker.sourceType = .camera
    imagePicker.delegate = self
    present(imagePicker, animated: true, completion: nil)
}
```

The previous implementation creates an instance of the `UIImagePickerController` object, and configures it so that it uses the camera as the image source and presents it to the user. Note that the view controller is set as a delegate for the image picker. When the user has taken a picture, the image picker will notify its delegate about this, so that it can extract the image and use it. In this case, the image should be given the `selectedImage` label in the view controller so it can be shown in the image view, and saved when the user taps on the save button and the `saveImage()` method is called as a result.

Add the following extension to make `ImageViewController` conform to `UIImagePickerControllerDelegate`:

```
extension ImageViewController: UIImagePickerControllerDelegate,
UINavigationControllerDelegate {
    func imagePickerController(_ picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {
        picker.dismiss(animated: true, completion: nil)

        guard let image = info[.originalImage] as? UIImage
        else { return }

        selectedImage = image
    }
}
```

```
    }  
}
```

Note that this extension also makes the image view controller conform to `UINavigationControllerDelegate`. The `delegate` property on the image picker controller requires all delegates to conform to both `UINavigationControllerDelegate` and `UIImagePickerControllerDelegate`.

When the user has taken a picture with the camera, `imagePickerController(_: didFinishPickingMediaWithInfo)` is called to notify the delegate about the photo that the user took. The first thing that the preceding code does is dismiss the picker, as it's no longer needed. The picture that the user just took is stored in the `info` dictionary as the original image. When the image is extracted from the dictionary, it is set as `selectedImage`.

To store the image, add the following implementation of `saveImage()`:

```
@IBAction func saveImage() {  
    guard let image = selectedImage  
    else { return }  
  
    UIImageWriteToSavedPhotosAlbum(image, self,  
#selector(didSaveImage(_:withError:contextInfo:)), nil)  
}  
  
@objc func didSaveImage(_ image: UIImage, withError error: Error?,  
contextInfo: UnsafeRawPointer) {  
    guard error == nil  
    else { return }  
  
    presentAlertWithTitle("Success", message: "Image was saved  
successfully")  
}
```

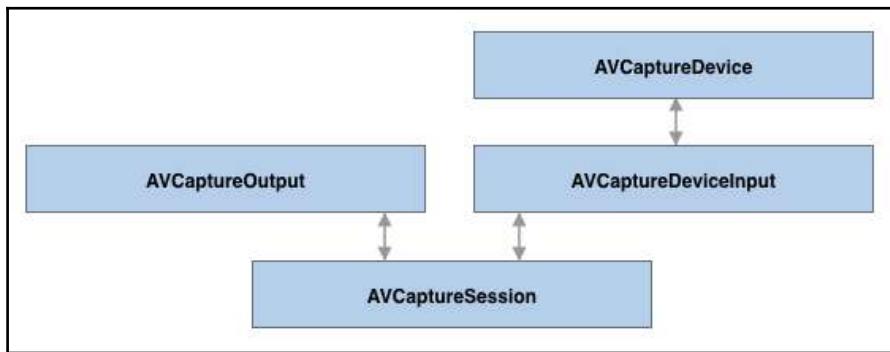
The preceding code calls `UIImageWriteToSavedPhotosAlbum(_:_:_:_)` to store the image in the user's photo library. When the save operation completes, the `didSaveImage(_:withError:contextInfo:)` method will be called. If this method does not receive any errors, then the photo was successfully stored in the photo library and an alert is shown.

Allowing the user to take a picture by implementing `UIImagePickerController` is relatively straightforward, and it's a great way to implement a camera feature in your app without too much effort. Sometimes, you need more advanced access to the camera. In these cases, you can use `AVFoundation` to gain access to the raw video feed from the camera, as you will see next.

Recording and storing video

In the previous section, you used AVFoundation to build a simple audio player app. You will now use AVFoundation again, except instead of playing video or audio, you will now record video and store it in the user's photo library. When using AVFoundation to record a video feed, you do so with an `AVCaptureSession` object. A capture session is responsible for taking the input from one or more `AVCaptureDeviceInput` objects, and writing it to an `AVCaptureOutput` subclass.

The following diagram shows the objects that are involved with recording media through an `AVCaptureSession`:



To get started on implementing the video recorder, make sure to import `AVFoundation` in `RecordVideoViewController.swift`. Also, add the following properties to the `RecordVideoViewController` class:

```
let videoCaptureSession = AVCaptureSession()  
let videoOutput = AVCaptureMovieFileOutput()  
  
var previewLayer: AVCaptureVideoPreviewLayer?
```

Most of the preceding properties should look familiar, because they were also shown in the screenshot that outlined the components that are involved with an `AVCaptureSession`. Note that `AVCaptureMovieFileOutput` is a subclass of `AVCaptureOutput`, specialized in capturing video. The preview layer will be used to render the video feed at runtime and present it to the user, so they can see what they are capturing with the camera.

The next step is to set up the `AVCaptureDevice` objects for the camera and microphone and associate them with the `AVCaptureSession`. Add the following code to the `viewDidLoad()` method:

```
// 1
guard let camera = AVCaptureDevice.default(.builtInWideAngleCamera,
for: .video, position: .back),
      let microphone = AVCaptureDevice.default(.builtInMicrophone, for:
.audio, position: .unspecified)
else { return }

// 2
do {
    let cameraInput = try AVCaptureDeviceInput(device: camera)
    let microphoneInput = try AVCaptureDeviceInput(device: microphone)

    videoCaptureSession.addInput(cameraInput)
    videoCaptureSession.addInput(microphoneInput)
    videoCaptureSession.addOutput(videoOutput)
} catch {
    print(error.localizedDescription)
}
```

The preceding code first obtains a reference to the camera and microphone that will be used to record the video and audio. The second step is to create the `AVCaptureDeviceInput` objects that are associated with the camera and microphone and associate them with the capture session. The video output is also added to the video capture session. If you examine the screenshot that you saw earlier and compare it with the preceding code snippet, you will find that all four components are present in this implementation.

The next step is to provide the user with a view that shows the current camera feed, so they can see what they are recording. Add the following code to `viewDidLoad()` after the capture session setup code:

```
previewLayer = AVCaptureVideoPreviewLayer(session:
videoCaptureSession)
previewLayer?.videoGravity = .resizeAspectFill
videoView.layer.addSublayer(previewLayer!)

videoCaptureSession.startRunning()
```

The preceding code sets up the preview layer and associates it with the video capture session. The preview layer will directly use the capture session to render the camera feed. The capture session is then started. This does not mean that the recording session starts; rather, only that the capture session will begin processing the data from its camera and microphone inputs.

The preview layer is added to the view at this point, but it doesn't cover the video view yet. Add the following implementation for `viewDidLayoutSubviews()` to `RecordVideoViewController` to set the preview layer's size and position, so it matches the size and position of `videoView`:

```
override func viewDidLayoutSubviews() {
    super.viewDidLayoutSubviews()

    previewLayer?.bounds.size = videoView.frame.size
    previewLayer?.position = CGPoint(x: videoView.frame.midX, y:
        videoView.frame.size.height / 2)
}
```

Running the app now will already show you the camera feed. However, tapping the record button doesn't work yet, because you haven't yet implemented the `startStopRecording()` method. Add the following implementation for this method:

```
@IBAction func startStopRecording() {
    // 1
    if videoOutput.isRecording {
        videoOutput.stopRecording()
    } else {
        // 2
        guard let path = FileManager.default.urls(for: .documentDirectory,
            in: .userDomainMask).first
            else { return }

        let fileUrl = path.appendingPathComponent("recording.mov")

        // 3
        try? FileManager.default.removeItem(at: fileUrl)

        // 4
        videoOutput.startRecording(to: fileUrl, recordingDelegate: self)
    }
}
```

Let's go over the preceding snippet step by step to see what exactly is going on:

1. First, the `isRecording` property for the video output is checked. If a recording is currently active, the recording should be stopped.
2. If no recording is currently active, a new path is created to store the video temporarily.
3. Since the video output can't overwrite an existing file, the `FileManager` object should attempt to remove any existing files at the temporary video file path.
4. The video output will start recording to the temporary file. The view controller itself is passed as a delegate to be notified when the recording has begun and is stopped.

Since `RecordVideoViewController` does not conform to `AVCaptureFileOutputRecordingDelegate` yet, you should add the following extension to add conformance to `AVCaptureFileOutputRecordingDelegate`:

```
extension RecordVideoViewController:  
AVCaptureFileOutputRecordingDelegate {  
    // 1  
    func fileOutput(_ output: AVCaptureFileOutput, didStartRecordingTo  
fileURL: URL, from connections: [AVCaptureConnection]) {  
        startStopButton.setTitle("Stop Recording", for: .normal)  
    }  
  
    // 2  
    func fileOutput(_ output: AVCaptureFileOutput, didFinishRecordingTo  
outputFileURL: URL, from connections: [AVCaptureConnection], error:  
Error?) {  
        guard error == nil  
        else { return }  
  
        UISaveVideoAtPathToSavedPhotosAlbum(outputFileURL.path, self,  
#selector(didSaveVideo(at:withError:contextInfo:)), nil)  
    }  
  
    // 3  
    @objc func didSaveVideo(at path: String, withError error: Error?,  
contextInfo: UnsafeRawPointer?) {  
        guard error == nil  
        else { return }  
  
        presentAlertWithTitle("Success", message: "Video was saved  
successfully")  
        startStopButton.setTitle("Start Recording", for: .normal)  
    }  
}
```

3

You can now run the app and record some videos! Even though you have done a lot of manual work by implementing the video recording logic directly with an `AVCaptureSession`, most of the hard work is done inside of the `AVFoundation` framework. One final media-related feature to explore is applying visual filters to images using `Core Image`.

Manipulating photos with Core Image

In this chapter, you have already seen that iOS has powerful capabilities for recording and playing media. In this section, you will learn how you can manipulate images with **Core Image**. The **Core Image** framework provides many different filters that you can use to process both images and video. You will expand on the photo-taking capabilities that you implemented in the **Captured** app so users can grayscale and crop images.

Every Core Image filter you apply to images is an instance of the `CIFilter` class. You can create instances of filters as follows:

```
let filter = CIFilter(name: "CIPhotoEffectNoir")
```

The name parameter in the filter's initializer is expected to be a string that refers to a specific filter. You can refer to Apple's documentation on Core Image and the Core Image Filter Reference guide to see an overview of all the filters that you can use in your apps.

Every filter has a certain set of parameters that you need to set on the `CIFilter` instance to use the filter. For instance, a grayscale filter requires you to provide an input image. Other filters might take an intensity, location, or other properties. The best way to see how you can apply a filter to an image is through an example. Add the following implementation for `applyGrayScale()` to `ImageViewController.swift` to implement a grayscale filter:

```
@IBAction func applyGrayScale() {
    // 1
    guard let cgImage = selectedImage?.cgImage,
        // 2
        let initialOrientation = selectedImage?.imageOrientation,
        // 3
        let filter = CIFilter(name: "CIPhotoEffectNoir")
    else { return }

    // 4
    let sourceImage = CIImage(cgImage: cgImage)
    filter.setValue(sourceImage, forKey: kCIInputImageKey)

    // 5
    let context = CIContext(options: nil)
    guard let outputImage = filter.outputImage,
        let cgImageOut = context.createCGImage(outputImage, from:
    outputImage.extent)
    else { return }

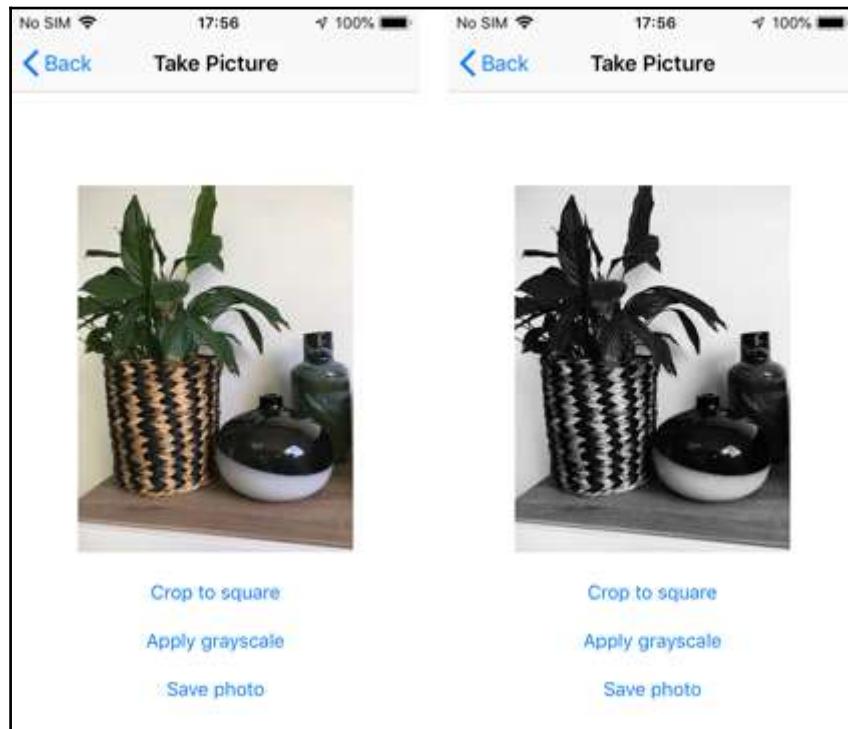
    // 6
    selectedImage = UIImage(cgImage: cgImageOut, scale: 1, orientation:
initialOrientation)
}
```

The preceding code has a lot of small, interesting details, highlighted with numbered comments. Let's go over the comments one by one to see how the grayscale filter is applied:

1. The `UIImage` instance that is stored in `selectedImage` is converted to a `CGImage` instance. Strictly speaking, this conversion isn't required, but it does make applying other filters to the `UIImage` instance later a bit easier.
2. One downside of using `CGImage`, instead of `UIImage`, is that the orientation information that is stored in the image is lost. To make sure the final image maintains its orientation, the initial orientation is stored.
3. This step creates an instance of the grayscale filter.

4. Since Core Image does not directly support `CGImage` instances, the `CGImage` instance is converted to a `CIImage` instance that can be used with Core Image. The `CIImage` instance is then assigned as the input image for the grayscale filter, by calling `setValue(_:forKey:)` on the filter.
5. The fifth step extracts the new image from the filter, and uses a `CIContext` object to export the `CIImage` output to a `CGImage` instance.
6. The sixth and final step is to create a new `UIImage` instance, based on the `CGImage` output. The initial orientation is passed to the new `UIImage` instance to make sure it has the same orientation as the original image.

Even though there are a lot of steps involved, and you need to convert between different image types quite a bit, applying the filter is relatively simple. Most of the preceding code takes care of switching between image types, while the filter itself is set up in just a couple of lines. Try running the app now and taking a picture. The initial picture will be in full color. After you apply the grayscale filter, the image is automatically replaced with a grayscale version of the image, as shown in the following screenshot:



The next filter you will implement is a crop filter. The crop filter will crop the image so that it's a square, rather than a portrait or landscape picture. The process for implementing the crop filter is mostly the same as the grayscale filter, except for the values that need to be passed to the crop filter. Add the following implementation for `cropSquare()` to implement the crop filter:

```
@IBAction func cropSquare() {
    let context = CIContext(options: nil)

    guard let cgImage = selectedImage?.cgImage,
          let initialOrientation = selectedImage?.imageOrientation,
          let filter = CIFilter(name: "CICrop")
    else { return }

    let size = CGFloat(min(cgImage.width, cgImage.height))
    let center = CGPoint(x: cgImage.width / 2, y: cgImage.height / 2)
    let origin = CGPoint(x: center.x - size / 2, y: center.y - size / 2)
    let cropRect = CGRect(origin: origin, size: CGSize(width: size,
                                                       height: size))

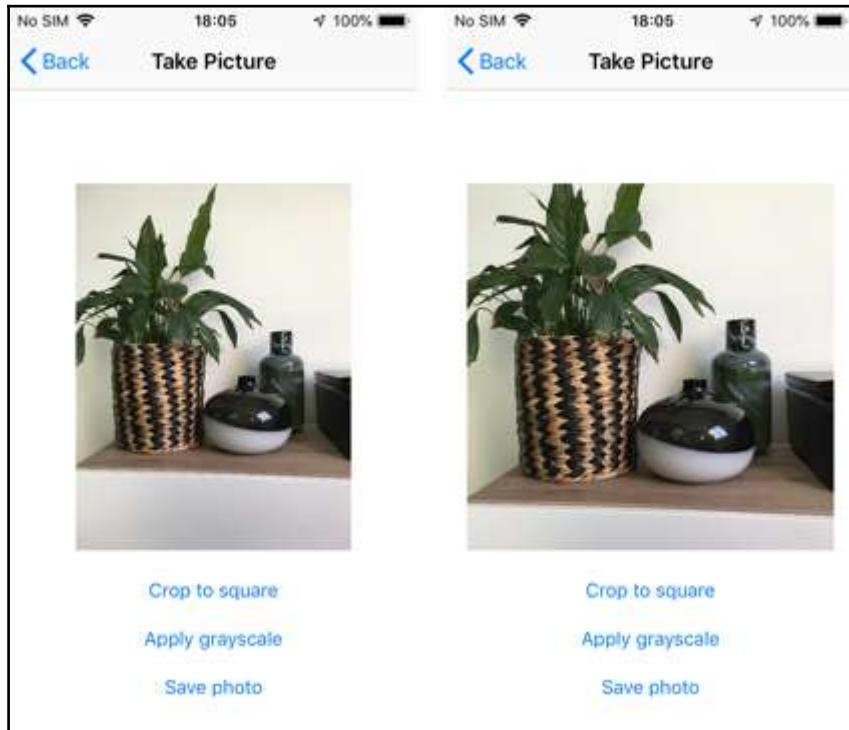
    let sourceImage = CIImage(cgImage: cgImage)
    filter.setValue(sourceImage, forKey: kCIInputImageKey)
    filter.setValue(CIVector(cgRect: cropRect), forKey:
        "inputRectangle")

    guard let outputImage = filter.outputImage,
          let cgImageOut = context.createCGImage(outputImage, from:
            outputImage.extent)
    else { return }

    selectedImage = UIImage(cgImage: cgImageOut, scale: 1, orientation:
        initialOrientation)
}
```

The preceding code performs several calculations to figure out the best way to crop the image into a square. The `CGRect` instance that specifies the crop coordinates and size, which are then used to create a `CIVector` object. This object is then passed to the filter as the value for the `inputRectangle` key. Apart from specifying the crop values, the process of applying the filter is identical, so the code should look familiar to you.

If you run the app now and tap the crop button, the image will be cropped, as shown in the following screenshot:



There are many more filters available in Core Image, which you can play with to build pretty advanced filters. You can even apply multiple filters to a single image to create elaborate effects for the pictures in your apps. Because all filters work in very similar ways, it's relatively easy to apply any filter to your images once you understand how the general process of applying a filter works. You can always use the code from the preceding examples if you need a reminder about how to apply Core Image filters.

Summary

In this chapter, you have learned a lot about media in iOS. You saw how you can implement a video player with just a couple of lines of code. After that, you learned how to use AVFoundation directly to build an audio player that supports features such as stopping and resuming playback, skipping songs, and scrubbing forward or backward in a song. You even learned how you can keep playing audio when the app goes to the background or when the phone is set to silent mode. To put the finishing touches to the audio player, you learned how you can use the MediaPlayer framework to show the currently playing file on the user's lock screen, and to respond to control events that are sent to the app remotely.

After implementing media playback, you learned how you can build apps that help users to create media. You saw that the UIImagePickerController provides a quick and simple interface to allow users to take a picture with the camera. You also learned how you can use AVFoundation and an AVCaptureSession object to implement a custom video recording experience. To wrap it all up, you learned about the Core Image framework, and how you can use it to apply filters to images.

In the next chapter, you will learn everything you need to know about notifications and how you can use them to make significant improvements to your apps.

Questions

1. Which framework is used to implement a simple video player?
 - a) AVFoundation
 - b) MediaPlayer
 - c) AVKit

2. Which object is used to play a local audio file?
 - a) AVPlayer
 - b) AVAudioPlayer
 - c) AVQueuePlayer

3. What are remote commands, and where do they come from?
 - a) Media playback-related commands that come from the lock screen.
 - b) Media playback-related commands that come from the control center.
 - c) Media playback-related commands that can come from any external source, including the lock screen, control center, or even the user's headphones.
4. Which object can be used to quickly take a picture?
 - a) UIImagePickerController
 - b) AVCaptureSession
 - c) AVCaptureDevice
5. Which of the following is not needed to record a video with an AVCaptureSession object?
 - a) A temporary file to store the video
 - b) Permission to access the user's photo library
 - c) An AVCaptureFileOutput object
6. What information tends to get lost when you convert a UIImage instance to a CGImage instance?
 - a) The orientation
 - b) The color profile
 - c) The previously applied filters

Further reading

- Big Buck Bunny download page: <http://bbb3d.renderfarming.net/download.html>
- The Free Music Archive: <https://freemusicarchive.org/about>
- Core Image Filter Reference: <https://developer.apple.com/library/archive/documentation/GraphicsImaging/Reference/CoreImageFilterReference>
- Core Image documentation: <https://developer.apple.com/documentation/coreimage>
- AVFoundation documentation: <https://developer.apple.com/av-foundation/>

22

Implementing Rich Notifications

In Chapter 20, *Streamlining Experiences With Siri*, you were introduced to app extensions by implementing an intent extension for Siri. You learned that app extensions allow you to integrate your app with iOS on a very deep level. Since iOS 10, Apple added extension capabilities for notifications, while also completely revamping how notifications work. In this chapter, you will learn how notifications work on iOS, and how you can implement extensions for notifications to create a rich notification experience.

First, you will learn about notifications in general. You will learn how notifications appear in iOS and how they work. Even though you won't dive straight into the fine details of implementing notifications, you will gain a deep understanding of notifications by looking at the different types of notifications, and the different ways of scheduling notifications that are available. Once you know how notifications work, you will learn how to implement and use them in your app.

By the end of this chapter, you will know everything there is to know about notifications and how they can be used to create a unique, engaging user experience. People receive lots of notifications throughout the day, so providing a great experience that stands out will ensure that people don't get annoyed with your notifications to the point of wanting to disable them entirely.

This chapter is divided into the following topics:

- Gaining a deep understanding of notifications
- Scheduling and handling notifications
- Implementing notification groups
- Implementing notification extensions

Gaining a deep understanding of notifications

If you use iOS devices on a daily basis, you are probably very familiar with notifications. Sometimes you may have been using your device when you received such a notification; other times, the device might have been in your pocket, on your desk, or somewhere else.

Notifications are a perfect way for apps to inform users about information in an app that might be of interest to them. New messages in a messaging app, breaking news events, or friend requests in social apps are just a few examples of great use cases for notifications. From the get-go, it's important that you're aware of the fact that a lot of apps are fighting for the user's attention at all times. Notifications are a great way to gain the user's attention, but if you send too many notifications, or only use notifications as a marketing tool, instead of as a way to provide meaningful information, it's very likely that the user is going to disable push notifications for your app at some point.

This is especially true since iOS 12. When a user swipes to ignore a notification, the user will be asked if they would like to adjust the notification settings for the app that sent the notification, or even turn off notifications for the app entirely. This new behavior makes it more important than ever to only send meaningful notifications to users.

The following six forms of notifications can be sent to users:

- Sounds
- Vibrations
- Visual notifications (such as banners and alerts)
- Badges
- Critical notifications
- Provisional

Most apps use a combination of these, and you will learn more about them in a real-world scenario soon. A unique type of notification is the critical notification. A *critical notification* is sent to the user regardless of whether or not the device is in *Do Not Disturb* mode, and plays sound even if the device is muted. Critical notifications can't be implemented by all applications, as they are reserved for apps that have very good use cases for sending potentially disruptive notifications.

Examples of good use cases are health-related notifications or alarm systems. If you plan to send critical notifications to your users, you must ask Apple for permission, so they can grant your app the appropriate entitlements if they decide that your use case is valid.

Sending provisional notifications is a new feature in iOS 12 that allows your app to send notifications silently to your users, without explicit permission beforehand. Asking your user for permission to send them provisional notifications will automatically grant your app permission to send notifications without ever prompting the user. Instead, the user will silently receive notifications from your app, and they will be able to choose whether they want to continue receiving notifications from your app, or deny the notifications permission altogether.

There are two ways for apps to send notifications to their users: local notifications and push notifications. **Local notifications** are scheduled by an app and kept on the device. To trigger these notifications, you use either a location, calendar, or time-based trigger.

Push notifications are pushed to a device through a server and the **Apple Push Notification Service (APNS)**. Push notifications are always delivered by APNS using best-effort delivery. This means that a notification will be sent to the user, and if this fails, APNS may or may not try again. There are no guarantees made about this, so you should never assume that your notification did or did not end up on the user's device.

Push notifications can send either a user-facing notification, or a silent notification. A user-facing notification will actively notify the user about what happened, often in the shape of a banner or badge icon. A silent notification signals the app that it should be woken up to perform a background fetch. You have seen this mechanism in action in Chapter 15, *Syncing Data with CloudKit*.

When a single app has multiple notifications to show to the user, iOS will group them automatically to avoid spamming the user's notification screen with a huge list of notifications from a single app.

The following screenshot shows a sample of grouped notifications:



While this type of grouping might be exactly what your app needs, there are cases where you might want to group notifications in a slightly different way. For instance, if you are building a chat application, you might want to group all messages for a single conversation, but show multiple notifications if a user has notifications for several ongoing conversations, as shown in the following screenshot:



When you schedule or send a notification, you can specify a thread identifier for that notification. If the thread identifier is set for a notification, iOS will group your notifications based on this identifier, rather than grouping all notifications for a single application together.

In order to complement the notifications you send to your user, you can implement **notification extensions**. These extensions add a whole new dimension to notifications, especially for devices that support 3D-Touch. There are two types of extensions you can add to a notification—service extensions and content extensions.

Service extensions allow you to modify or augment the contents of a notification before it is shown to the user. **Content extensions** allow you to display a custom notification interface when the user 3D-Touches a notification.

You will learn more about notification grouping and extensions later. Before that, you will learn how to ask the user for permission to send them notifications, and how to schedule and handle notifications.

Scheduling and handling notifications

Every notification you see on iOS has been scheduled or pushed to the user in one way or another. But before an app is even allowed to send notifications, there are a couple of things you must do. The steps you need to follow can be roughly divided as follows:

1. Registering for notifications
2. Creating notification contents
3. Scheduling your notification
4. Handling incoming notifications

Not every step is completely identical for local and remote notifications. When there are differences, and these will be very minor, they will be described in the relevant section. You will work on the **Notifications** app that is included in the code bundle for this chapter.

The Notifications app contains a user interface that allows you to schedule different notifications for reminders that will trigger after a certain amount of time, so you can easily experiment with different notifications.

Registering for notifications

Any app that wants to present notifications has to ask the user for permission before they do so. This process is called registering for notifications. It's important that you make sure your user understands why you're asking their permission to send notifications. After all, you're interrupting your user while they're trying to achieve something in your app, and the permission alert prevents them from doing so. It's often a great idea to hold off on asking for permission to send notifications until the user does something that can be improved specifically by turning on notifications.

In the Notifications example app, a user can add reminders for certain actions that they need to perform. The best time to ask a user for permission to send them notifications is right after they create a new reminder. This will make it clear to the user that you want to send notifications to inform the user about the reminder they just added.

When you ask for permission to send notifications, you can specify the kinds of notification that you want to send the user. You can pick from the following types:

- Alerts
- Badges
- CarPlay
- Sounds
- Critical
- Provisional

The user can go into your app's page in the *Settings* app at any time to change these settings. You can read these settings with the `UserNotifications` framework. By reading the current notification settings, you can inform your users about the current notification status, and ask them to turn notifications on if needed.

When you ask for a user's permission to send notifications and they deny permission to do so, or they disable notifications at a later stage, it's often a good idea to notify the user that they have notifications turned off, and then provide an easy way to turn notifications back on.

Reading the current notification settings and asking for permission to send notifications is always done through an `UNUserNotificationCenter` instance. You will typically obtain an instance of `UNUserNotificationCenter` through the static `current()` method.

`UNUserNotificationCenter` is not to be confused with `NotificationCenter`. `UNUserNotificationCenter` is responsible for notifications such as push and local notifications, while `NotificationCenter` is used to send notifications internally inside your app. The user never sees or notices these notifications.

As you have learned before, it's good practice to extract certain parts of your code into a dedicated helper object. Create a new helper folder in the **Notifications** project and add a new Swift file, called `NotificationsHelper.swift`, to the group. Add the following implementation code to your new helper file:

```
import UIKit
import UserNotifications

struct NotificationsHelper {
    let notificationCenter = UNUserNotificationCenter.current()
}
```

The best moment to ask a user for permission to send them push notifications for this app is when they have just added a reminder. At this point, the user will understand exactly why they see the permission dialog, because they just performed an action that is closely related to the push notifications they will receive.

Add the following method to the notifications helper to ask the user for permission to send them notifications:

```
func requestNotificationPermissions(_ completion: @escaping (Bool) -> ())
{
    notificationCenter.requestAuthorization(options: [.badge, .sound,
.alert]) { permissionGranted, error in
        completion(permissionGranted)
    }
}
```

The preceding method calls the notification center's `requestAuthorization(options:completion:)`, and calls the completion closure that was passed to `requestNotificationPermissions(_:)` with the result of the permission prompt.

Before you can use the helper in the add notifications view controller to ask the user for notification permissions when they add a new reminder, you need to create an instance of the helper in `AddNotificationViewController`. Add the following property to this class:

```
let notificationsHelper = NotificationsHelper()
```

Also, add the following code to the end of the `createReminder(_:withComponents:inContext:)` method to ask the user for notification permissions after creating a new reminder:

```
DispatchQueue.main.async { [weak self] in
    self?.notificationsHelper.requestNotificationPermissions { result in
        DispatchQueue.main.async {
            self?.enableNotificationsButton.isHidden = result
        }
    }
}
```

Since the code inside of the `createReminder(_:withComponents:inContext:)` method is executed on a managed object context's queue, it's good practice to ensure that the notification prompt will be presented using the main thread, by wrapping the code in a `DispatchQueue.main.async` call. To learn more about how dispatch queues work and how you can use them, refer to Chapter 28, *Offloading Tasks with Operations and GCD*.

In addition to asking for notification permissions when the user adds a new reminder, permission should also be asked when the user enables the periodic drink water notification. Add the following implementation for

`drinkWaterNotificationToggled(sender:)`:

```
@IBAction func drinkWaterNotificationToggled(sender: UISwitch) {
    if sender.isOn {
        notificationsHelper.requestNotificationPermissions { [weak self]
            result in
            DispatchQueue.main.async {
                self?.enableNotificationsButton.isHidden = result
            }
        }
    }
}
```

The add notification view contains a button that prompts the user to enable notifications for the app. This button should only be visible if the user has denied the notification permissions dialog, or if the user has turned off notifications for the app in their settings. Add the following method to the notifications helper to determine whether the user has turned off notifications for the app:

```
func userHasDisabledNotifications(_ completion: @escaping (Bool) ->
() {
    notificationCenter.getNotificationSettings { settings in
        completion(settings.authorizationStatus == .denied)
    }
}
```

The preceding code retrieves the current notification settings, and checks whether the current status is `.disabled`. There are a total of four possible authorization statuses: `.authorized`, `.denied`, `.notDetermined`, and `.provisional`. The following list describes what each status means:

- `notDetermined` means that the app hasn't asked for permissions before.
- `authorized` means that the app has asked for permissions and is allowed to send notifications to the user.
- `denied` means that the app has asked for permissions to send notifications to the user, and the user denied this permission, because they either declined the permissions prompt, or they turned off notifications in their settings.
- `provisional` means that the app is allowed to deliver quiet notifications to the app on a provisional basis.

It's a good idea to read the current notification settings whenever the add notification view appears, so the button that takes the user to the settings page is always shown and hidden correctly, based on the current notification permissions. Add the following implementation for `viewWillAppear(_:)` to the

`AddNotificationViewController` class:

```
notificationsHelper.userHasDisabledNotifications { [weak self]
    notificationsDisabled in
    DispatchQueue.main.async {
        self?.enableNotificationsButton.isHidden = !notificationsDisabled
    }
}
```

Before you test the app, add the following implementation for the `openSettingTapped()` method.

Add the following implementation for the `enableNotificationsTapped` action:

```
@IBAction func openSettingsTapped() {
    let settingsUrl = URL(string: UIApplication.openSettingsURLString)
    UIApplication.shared.open(settingsUrl!, options: [:],
    completionHandler: nil)
}
```

The preceding implementation will take the user to the *Settings* app, so they can easily enable notifications for the Notifications app. If you test the app now, you can add a new reminder, and the app will ask you for permission to send notifications. If permission is denied, the *enable* button will appear at the bottom of the view. When you tap this button, you will be taken directly to the setting page, where you can again enable notifications.

The preceding code registers the app for local notifications. If you want to register the app for push notifications, you must enable the **Push Notifications** capability in the **Capabilities** tab. Note that you can only enable this capability if you're on Apple's paid developer program. Free developer accounts cannot enable this capability.

After enabling the push notifications capability, you need to add a couple of lines of code to make sure that you properly register the current device with APNS. To do this, you need to call a method on `UIApplication` that registers the current device on APNS. After calling this method, a delegate method in `AppDelegate` is called with the current device token that should be uploaded to your remote server, so you can send messages to the device through the device token.

Apple makes no guarantees about if and when a device's push token can change, so you should always register for remote notifications whenever the app launches. For the Notifications app, it makes sense to register for remote notifications as soon as the app enters the foreground, or when the user has just granted the app permission to send push notifications.

Add the following lines of code to the `requestNotificationPermissions(_:)` method in the notification helper, right after calling the completion handler:

```
DispatchQueue.main.async {  
    UIApplication.shared.registerForRemoteNotifications()  
}
```

Note that you must always call `UIApplication.shared.registerForRemoteNotifications()` on the main thread. Since the callback for `requestNotificationPermissions(_:)` is not guaranteed to be called on the main thread, you should manually ensure that you register for remote notifications on the main thread.

Next, implement the following methods in `AppDelegate`:

```
func applicationDidBecomeActive(_ application: UIApplication) {  
    UIApplication.shared.registerForRemoteNotifications()  
}
```

```
func application(_ application: UIApplication,  
didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {  
    print("received device token: \(deviceToken)")  
}  
  
func application(_ application: UIApplication,  
didFailToRegisterForRemoteNotificationsWithError error: Error) {  
    print("Did not register for remote notifications:  
\((error.localizedDescription)")  
}
```

For demo purposes, the device token is printed to the console. In a real app, you should upload the device token to your server instead. You're all set to send notifications to your users now. Next, let's look at how you can create and configure notifications in your app.

Creating notification contents

Notifications typically consist of a few pieces of content. First of all, every notification has a title, a subtitle, and a message body. Not all fields have to be present, but you'll often want to provide a bit more information than just a piece of body text. You can also add media attachments to notifications. These attachments are used to provide a preview of a photo, video, or audio file that is stored locally on the device, and are great for providing better-looking, richer notifications to your user.

The biggest gap between local and remote notifications exists in the way you create them. Push notifications are always created on the server and are pushed to the device in a JSON format. Local notifications are created on the device using the `UNNotification` class. Let's look at each notification type separately.

Creating push notifications

If you ignore the complexities of setting up and hosting your own server, push notifications are actually pretty simple to create. Push notifications are always delivered using a JSON payload that your own server sends to APNS. APNS will then take care of sending your notification to the device that's intended to receive it. Let's look at a simple example of a JSON payload that could be used to inform users about a newsworthy event:

```
{  
    "aps": {  
        "alert": {  
            "title" : "Breaking news!",  
            "body" : "iOS 12 was released"  
        },  
        "badge" : 1  
    },  
    "item_id" : "1"  
}
```

This simple notification contains only a title and a body. The number 1 is added to the app icon to indicate the number of news items available. You'll notice that the notification content is placed in the `aps` dictionary. The `aps` dictionary contains the push notification's content. The `item_id` is not part of the `aps` dictionary on purpose; it's a custom property that can be read in the app so it can retrieve data about the news item from a local database, or fetch it from a backend using the ID.

If you want to send a silent push notification to trigger a background data refresh action, like CloudKit does, you should add the `content-available` key to the `aps` dictionary, specifying a value of 1. If you want to use a custom sound for your notification, you can add a `sound` key to `aps`. The value of this key should point to an audio file that's embedded in your app. Finally, a notification can be part of a category of notifications. If this is the case, add a `category` key to the `aps` dictionary, where the value is the category identifier. For an overview of all of the possible keys that can be added to the `aps` dictionary, have a look at Apple's Payload Key Reference documentation.

Creating local notifications

Local notifications are created and stored locally on the device. They are very similar to push notifications in terms of the content they can contain. The way that content is created is quite different though. For instance, you can't create a local notification for content that doesn't exist yet.

In the context of a news app, this means that a server could push a news item to the application when needed. If you create the notification locally, you don't know what tomorrow's big news item will be, so you can't schedule a notification with content for this news item yet. This is an important distinction between local and push notifications that you should keep in mind when choosing whether you should create a notification locally, or if you should leave it up to a server to create the notification.

The following code snippet demonstrates the creation of a notification for a reminder. You should add this snippet to the `NotificationsHelper`:

```
func createNotificationContentForReminder(_ reminder: Reminder) ->
UNMutableNotificationContent {
    let content = UNMutableNotificationContent()

    content.title = "Reminder"
    content.body = reminder.title ?? ""
    content.badge = 1

    content.userInfo = ["reminder-uuid":
reminder.identifier?.uuidString]

    return content
}
```

This preceding code contains data that is quite similar to the JSON you saw previously. The main difference is that you create an instance of `UNMutableNotificationContent` and assign values to it, instead of defining the entire notification using JSON. A lot of the options that are available in the push example are also available for local notifications. You can add custom properties, such as the unique identifier for a reminder, to a `UNNotification` object's `userInfo` dictionary. You can also assign a custom sound, attachment, and more to local notifications, just as you can for push notifications.

Now that you know how to create notification contents, let's see how you can schedule them.

Scheduling your notification

In order to deliver local notifications to your users, you'll need to schedule them. This is another difference between local and push notifications. Push notifications do not need to be scheduled, because they are pushed through the server and delivered as soon as possible. When you schedule a local notification, you make use of a notification trigger. There are several notification triggers that you can use to send a notification:

- Location-based
- Calendar-based
- Time-based

Before you implement a proper notification to send reminders to a user, you will create a notification that is scheduled to fire a couple of seconds after creating it. You will then learn how to send a repeating notification that uses a time interval. Lastly, you will learn about calendar-based and location-based triggers.

Scheduling a timed notification

A timed notification is the simplest notification to schedule, which makes it a great candidate to explore first. Timed notifications are scheduled with an instance of `UNTimeIntervalNotificationTrigger`. The initializer for this class takes a time interval and a boolean value to indicate whether the notification should be sent to the user repeatedly. Add the following method to the `NotificationHelper`:

```
func scheduleTimedNotificationWithContent(_ content:  
UNNotificationContent) {  
    let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10,  
repeats: false)  
    let request = UNNotificationRequest(identifier: UUID().uuidString,  
content: content, trigger: trigger)  
    notificationCenter.add(request) { error in  
        if let error = error {  
            print(error.localizedDescription)  
        }  
    }  
}
```

The preceding method creates a trigger for the notification. In this case, the notification is set up to fire after 10 seconds have passed. Next, a notification request is created. Every notification request should have an identifier. In this case, a very simple identifier is used to identify the timed notification. After creating the notification, it is added to the notification with a completion handler.

The `identifier` attribute for a notification should be unique for the content of the notification. The system uses this identifier to avoid sending multiple notifications with the same content. If you send multiple notifications with the same identifier, the last notification you sent will be shown to the user. If you want to display multiple notifications in the Notification Center, you should make sure that every notification you schedule has a unique identifier.

If you want to send a repeating notification to a user, you must set the `repeats` property for the time interval trigger to `true`.



The minimum time interval for repeating notification is 60 seconds. Scheduling a repeating trigger with an interval under 60 seconds will crash your application.

To test the interval-based notification, you can add a call to `scheduleTimedNotificationWithContent (_:)` in `AddNotificationViewController` in the `addBedtimeNotification()` or `addLunchTimeNotification()` methods, after the following line:

```
try? self.persistentContainer.viewContext.save()
```

The following code can be added to schedule the timed notification:

```
let content =  
    notificationsHelper.createNotificationContentForReminder(reminder)  
    notificationsHelper.scheduleTimedNotificationWithContent(content)
```

If you run the app now, and schedule a reminder before going to your device's home screen or the lock screen, a notification about the reminder should appear after about 10 seconds. The following screenshot shows an example of a lunchtime reminder:



The next notification you will explore is the calendar based notification.

Scheduling a calendar-based notification

Calendar-based notifications make use of DateComponents to determine when they should be fired. The only difference compared to the time-interval scheduling is in the trigger that is used to determine when the notification should be displayed.

Go right ahead and add the following method to NotificationsHelper to create a notification that is fired every hour, on the hour:

```
func scheduleHourlyWaterReminder() {
    let content = UNMutableNotificationContent()

    content.title = "Reminder"
    content.body = "This is your reminder to drink some water every
hour."

    var hourComponents = DateComponents()
    hourComponents.minute = 0

    let trigger = UNCalendarNotificationTrigger(dateMatching:
hourComponents, repeats: true)
    let request = UNNotificationRequest(identifier: "water-
notification", content: content, trigger: trigger)

    notificationCenter.add(request) { error in
        if let error = error {
            print(error.localizedDescription)
        }
    }
}
```

The preceding code sets up a trigger that fires every hour when the value of the minutes component is 0. Since this notification should trigger every hour, the repeats argument is set to true.

Since the user can choose to turn off the water reminder, you should also implement a method to remove the repeating notification. Add the following method to the notifications helper to do this:

```
func unscheduleHourlyWaterReminder() {
    notificationCenter.removePendingNotificationRequests(withIdentifiers:
["water-notification"])
}
```

Next, update `drinkWaterNotificationToggled(sender:)` in the add notifications view controller, so that `scheduleHourlyWaterReminder()` is called when the switch is on and `unscheduleHourlyWaterReminder()` is called when the switch is off.

Calendar-based notifications provide a powerful way for you to schedule both recurring and one-time-only notifications that are tied to a specific moment in time that's easier to express in days, hours, weeks, or months, than in a time interval.

Scheduling a location-based notification

The last type of notification you can schedule is a location-based notification. To set up a location-based trigger, you use Core Location to set up a region to which the trigger should belong.

A location-based trigger is actually very similar to implementing geofencing. If you need a refresher on geofences, skip back to Chapter 17, *Improving Apps With Location Services*. The main difference between a location-based notification and a geofence is that when a geofence triggers, the user is not necessarily informed about this. If a location-based notification triggers, the intent is to inform the user about it so they see a notification. Let's look at an example of how a location-based notification is set up:

```
let coords = CLLocationCoordinate2D(latitude: 52.373095, longitude:  
4.8909129)  
let region = CLCircularRegion(center: coords, radius: 1000,  
identifier: "quote-notification-region")  
region.notifyOnExit = false  
region.notifyOnEntry = true  
let trigger = UNLocationNotificationTrigger(region: region, repeats:  
false)
```

The preceding snippet sets up a location-based notification, using a circular region with a radius of 1000 meters. The `notifyOnExit` and `notifyOnEntry` properties are used to specify whether the user should be notified when they enter the region, exit it, or when either event occurs.

Similar to the other notification triggers, you can choose whether the notification should be fired repeatedly, or just once.

This wraps up the exploration of all the ways for you to schedule notifications. Let's explore the last piece of the puzzle of notifications: handling incoming notifications and notification actions.

Handling notifications

When a user receives a notification, it's typically displayed on the lock screen and in the Notification Center. The user can see the contents of the notification, and they have the option to tap the notification to launch the app that the notification belongs to.

In addition to tapping on a notification, users can swipe a notification to make extra buttons appear, which enable them to quickly perform an action that is related to the notification; for instance, accepting a calendar invite or quickly replying to a message. In this section, you will explore the following aspects of notification handling:

- Handling notifications in your app
- Managing pending and delivered notifications
- Adding actions to notifications

Properly implementing the preceding aspects can greatly enhance the value of the notifications that you send to your users.

Handling notifications in your app

Incoming notifications are most often received when the user is not using your app. This means that your app is not directly aware of the incoming notification, and doesn't get to handle it until the user taps the notification to open your application.

When your application launches due to a tapped notification, the `UNUserNotificationCenterDelegate` delegate protocol that belongs to the `UNUserNotificationCenter` object is called. This delegate is responsible for handling the incoming notification whether the app is currently in the foreground, background, or not active at all, perhaps because it's been killed by the system or force-closed by the user.

You are expected to assign a `UNUserNotificationCenterDelegate` delegate before the app has finished launching. This means that you should set the delegate as soon as possible, so `application(_:didFinishLaunchingWithOptions:)` is a great candidate for setting the delegate.

When the user has tapped a notification, or selects one of the custom actions that is attached to the notification, the

`userNotificationCenter(_:didReceive:withCompletionHandler:)` method is called on the notification center delegate. If the app is in the foreground and a notification is received,

`userNotificationCenter(_:willPresent:withCompletionHandler:)` is called right before the notification is shown to the user. You can use this method to determine whether the notification should be shown to the user. Sometimes, it might not make sense to present a certain notification to the user if the app is already running. For instance, you wouldn't show a notification for an incoming message if the user is already looking at the new message.

Let's see what the notification center delegate methods look like when they are implemented. First, import the `UserNotifications` framework in `AppDelegate` and add the following code to implementation for `application(_: didFinishLaunchingWithOptions:)` to set the app delegate as a delegate for the `UNUserNotificationCenter`:

```
func application(_ application: UIApplication,  
didFinishLaunchingWithOptions launchOptions:  
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {  
  
    // existing code...  
  
    UNUserNotificationCenter.current().delegate = self  
  
    return true  
}
```

Next, add the following extension to `AppDelegate.swift` to make the app delegate conform to `UNUserNotificationCenterDelegate`:

```
extension AppDelegate: UNUserNotificationCenterDelegate {  
    func userNotificationCenter(_ center: UNUserNotificationCenter,  
    didReceive response: UNNotificationResponse, withCompletionHandler  
    completionHandler: @escaping () -> Void) {  
        let notification = response.notification  
        let action = response.actionIdentifier  
  
        let notificationTitle = notification.request.content.title  
        let customAttributes = notification.request.content.userInfo  
  
        completionHandler()  
    }  
}
```

```
func userNotificationCenter(_ center: UNUserNotificationCenter,  
willPresent notification: UNNotification, withCompletionHandler  
completionHandler: @escaping (UNNotificationPresentationOptions) ->  
Void) {  
    completionHandler([.alert, .sound])  
}  
}
```

Both delegate methods have access to the original notifications and the notification contents. You can use this information to determine exactly which actions should be taken by your app to properly handle the notification. When a notification is tapped, `userNotificationCenter(_:didReceive:withCompletionHandler:)` is called. This method receives a `UNNotificationResponse` object, which provides information about the selected action, in addition to the notification information.

If your app receives a notification and it's in the foreground, `userNotificationCenter(_:willPresent:withCompletionHandler:)` allows you to determine what should happen with the notification. This method is passed a closure that you need to call to inform iOS about what should happen with the notification. If you want to handle the notification yourself, you can simply call the closure without any arguments. In the preceding snippet, the desired behavior is to display all notification alerts to the user and play the corresponding sounds.

Because you can access the original notification, you can determine whether the notification should be shown for each incoming notification individually, depending on the notification content, category, or custom attributes.

Managing pending and delivered notifications

When you learned about scheduling notifications, you saw that each notification is assigned a unique identifier. If a new notification is added with an identifier that has already been used, the new notification will overwrite the existing notification, limiting the amount of similar and duplicate notifications in the Notification Center.

This overwrite behavior applies to both delivered and pending notifications. This means that you can replace a notification that the user has already received by scheduling a second notification with the same identifier. This is especially useful if the new notification makes the old notification obsolete. For instance, consider a situation when you send push messages to update a user about the current score for a sports match. If you push each score update with the same identifier, the user will always see a single notification with the most recent score.

You can also remove delivered notifications by calling `removeDeliveredNotifications(withIdentifiers:)` on the notification center in your app. If your app allows the user to select the types of notification they want to receive, you could use this feature to remove all notifications of a certain type. That way, the user won't see the notifications anymore.

To update notifications that have been sent by your server, you push a new notification with the `apns-collapse-id` header set on the request that your server sends to APNS. If this header is present, the new notification will overwrite any existing notifications that have been pushed with the same `apns-collapse-id`.

Updating your notifications, instead of pushing new ones all the time, is a great way to avoid needless clutter in the Notification Center. More importantly, there won't be any duplicate, redundant, or outdated messages fighting for the user's attention.

Let's see how you can further improve the notification experience through custom actions.

Adding actions to notifications

When you implemented the code to handle incoming notifications, you implemented a method called

`userNotificationCenter(_:didReceive:withCompletionHandler:)`. This method contains information about the way the user chose to handle the notification. They could have tapped on the notification itself, which is the default action, or maybe they chose one of the custom actions that are associated with the notification by swiping it. Another way for users to make custom notifications appear is to 3D-Touch on it.

There are three main kinds of actions supported by notifications. The first is a background action. Background actions dismiss the notification and wake the app in the background, so it can take appropriate measures for the selected action. An example of this is accepting a calendar invite through an action. If a user accepts or declines an invite inside the notification, they don't have to be taken to the app, because the response can be handled in the background just fine.

The second type of action is a foreground action. A foreground action takes the user into the app that sent the notification. It's up to the app to handle the selected action in a way that makes sense.

Lastly, you can add text input actions to your notifications. These types of actions are basically background actions, except they take user input. A text input action is great if you're building a messaging app.

All notification actions are associated with a notification category. These categories must be registered on the Notification Center at app launch. Again, the perfect place to set this up is `application(_:didFinishLaunchingWithOptions:)`. Add the following code to this method to configure a notification action that allows a user to mark one of their reminders as completed:

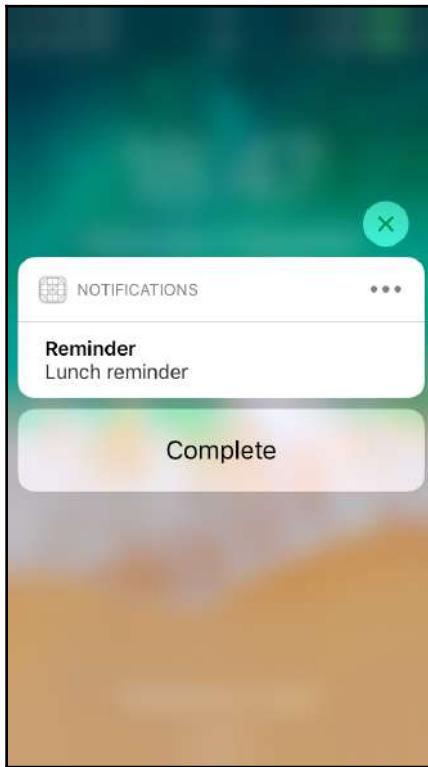
```
let completeAction = UNNotificationAction(identifier: "complete-  
reminder", title: "Complete", options: [])  
  
let reminderCategory = UNNotificationCategory(identifier: "reminder",  
actions: [completeAction], intentIdentifiers: [], options: [])  
  
UNUserNotificationCenter.current().setNotificationCategories([reminder  
Category])
```

The preceding code creates a custom action that is attached to the `reminder` category. The default behavior for a notification is to handle actions in the background. If you want an action to take the user into your app, you need to add the `.foreground` option to the `options` parameter for the action. Lastly, the new category is added to the Notification Center.

Associating a category with a notification is done through the `categoryIdentifier` property of `UNNotificationContent`. Add the following line of code to the `createNotificationContentForReminder(_:)` method in `NotificationsHelper.swift` to add the `reminder` category to the notifications that are scheduled by the app:

```
content.categoryIdentifier = "reminder"
```

If you 3D-Touch the notifications that is sent by the app now, you're presented with a custom action:



When the user taps on this action, `userNotificationCenter(_:didReceive:)` is called to notify your app about the fact that the user tapped on your custom action. Add the following implementation for this method to `AppDelegate.swift` to mark a reminder as completed when the user selects the **Complete** action:

```
func userNotificationCenter(_ center: UNUserNotificationCenter,  
didReceive response: UNNotificationResponse, withCompletionHandler  
completionHandler: @escaping () -> Void) {  
  
    let userInfo = response.notification.request.content.userInfo  
  
    guard response.actionIdentifier == "complete-reminder",  
        let identifier = userInfo["reminder-uuid"] as? String else {  
        completionHandler()  
        return  
    }  
}
```

```
let fetchRequest: NSFetchedResultsController<Reminder> = Reminder.fetchRequest()
fetchRequest.predicate = NSPredicate(format: "identifier == %@", identifier)

let moc = PersistentHelper.persistentContainer.viewContext
guard let results = try? moc.fetch(fetchRequest),
      let reminder = results.first else {
    completionHandler()
    return
}

moc.perform {
    reminder.isCompleted = true
    try! moc.save()

    completionHandler()
}
}
```

The preceding code extracts all relevant information from the notification response and uses it to find a reminder that matches the identifier that was added to the notification's `userInfo` dictionary. If a reminder was found, it's marked as completed and the managed object context is saved to persist the new reminder status.

Notification actions are a great way to add quick interactions to your notifications, and they are already a huge improvement over simply sending a plain notification. Another way to make great improvements to the way you send notifications to your users is to implement the new notification grouping that was introduced in iOS 12.

Implementing grouped notifications

A new feature for notifications in iOS 12 is the possibility to group notifications. iOS will group notifications from an application together by default, showing the user a stack of notifications with the most recent one on top, instead of a long list of mixed notifications from all applications, which is hard to scan through.

In this section, you will learn exactly how you can implement grouped notifications in your own apps.

Grouping notifications based on thread identifiers

Grouped notifications are a huge improvement for the notification system on iOS, which many users have eagerly been waiting for. However, if all notifications for a single app are grouped together, important information might get lost. If a messaging app stacks all notifications in a single group, it's easy to miss an important message if it's hidden somewhere in the group.

For this reason, Apple allows developers to create custom notification groups based on special thread identifiers. The most obvious use case is to group all notifications from a certain conversation in a messaging app together. However, there are other useful implementations of grouped notifications that you can come up with.

You could use thread identifiers to separate certain important, actionable notifications from a group of informative notifications. Doing this will make the important notifications stand out to the user, without cluttering the user's Notification Center with several less important informative messages.

To begin using grouped notifications, all you have to do is assign a thread identifier to a notification content object. For the Notifications app, it would be interesting to group notifications for bedtime and lunchtime notifications together. You can do this by assigning a thread identifier to the notification content object that is created by the notifications helper object. Update the implementation for `addLunchTimeNotification()` as follows to assign a thread identifier to a lunchtime notification:

```
@IBAction func addLunchTimeNotification() {
    persistentContainer.viewContext.perform { [unowned self] in
        // existing implementation ...
        let content =
        self.notificationsHelper.createNotificationContentForReminder(reminder)
        content.threadIdentifier = "lunchtime-thread"
        self.notificationsHelper.scheduleTimedNotificationWithContent(content)
    }
}
```

Only a single line of code was added to implement a thread identifier. Try adding a thread identifier to the bedtime notification as well, and then run the app. Try scheduling a couple of bedtime and lunchtime notifications to see how iOS groups the notifications based on the thread identifier.

Providing a custom summary message for your notification group

When iOS displays a grouped notification, a summary message is shown at the bottom of the notification bubble. By default, this message simply informs the user about the number of notifications that are available. You can customize this message by providing a summary format yourself.

Summary formats are tied to notifications through categories. This means that you can add different messages for different notification categories. A custom summary format is supplied in the form of a format string. Update the `UNNotificationCategoryIdentifier` initialization code in `AppDelegate.swift` to add a custom summary format for the reminder category as follows:

```
let summaryFormat = "%u more reminders for this topic."  
  
let reminderCategory = UNNotificationCategory(identifier: "reminder",  
actions: [completeAction], intentIdentifiers: [],  
hiddenPreviewsBodyPlaceholder: nil, categorySummaryFormat:  
summaryFormat, options: [])
```

The summary format passed to the `UNNotificationCategory` initializer uses the `%u` format specifier to specify where the number of reminders should be displayed. When you run the app and schedule a couple of notifications, the custom summary format should be displayed, as shown in the following screenshot:



Note that one of the notification groups shows a slightly strange summary message. It says **2 more reminders for this topic**. Note the plural for reminders being used in this summary message. It would be much nicer if the summary format could account for this and use "*1 more reminder for this topic*" instead, if there is only a single extra item in the notification group.

You can achieve this by implementing a localized string for the summary format. To do this, start by defining the summary format string in `AppDelegate` as follows:

```
let summaryFormat = NSLocalizedString.localizedUserNotificationString(forKey:  
    "REMINDER_SUMMARY", arguments: nil)
```

The preceding code defines the summary format as a localized string that is specific for notifications. When you define a string like this, iOS will look for the display string in a dedicated file containing all of the strings that you wish to localize for notifications, using the key you have specified.

Localizations for notifications use a special `.stringsdict` localization file that contains information that describes how to localize a string for several quantities. To add a `.stringsdict` file, go to **File | New** and select the `Stringsdict` file from the list of file types. Name your file `Localizable.stringsdict` and make sure it's added to your app target. Next, open the `Localizable.stringsdict` file and make sure its contents resemble the following screenshot:

Key	Type	Value
▼ Strings Dictionary	Dictionary	(1 item)
▼ REMINDER_SUMMARY	Dictionary	(2 items)
NSStringLocalizedFormatKey	String	%#@reminders@
▼ reminders	Dictionary	(4 items)
NSStringFormatSpecTypeKey	String	NSStringPluralRuleType
NSStringFormatValueTypeKey	String	u
one	String	%u more reminder for this topic.
other	String	%u more reminders for this topic.

The strings dictionary contains the key that was used in `AppDelegate` to identify the string that should be used for the summary label. The `NSStringLocalizedFormatKey` string contains a variable that is represented as `%#@reminders@`. This variable is the part of the string that you want to localize. In this case, the variable represents the entire message.

The variable name is then added as a key to the `REMINDER_SUMMARY` dictionary to specify the localization rules. In this case, a rule is added for a single item and for multiple items.



If you want to know more about the `stringsdict` file, make sure to read Apple's documentation on app localization.

Now that you know how to implement grouped notifications, let's see how you can add Notification Extensions to your app.

Implementing notification extensions

Apple has created two extension types for notifications that you can use to take the notification experience up a notch from custom actions. The available extensions are **service extensions** and **content extensions**. These extensions are both very powerful and relatively simple to implement. In this section, you will learn how to add both a service extension and a content extension to the Notifications app.

Adding a service extension to your app

Service extensions are intended to act as middleware for push notifications. A service extension receives a notification before it's displayed to the user. This allows you to manipulate or enrich the notification's content before it's shown to the user.

A service extension is perfect if you're implementing end-to-end encryption, for example. Another great use for a service extension is to download a media attachment from a push notification, save it locally, and add it as a media attachment to the notification contents, because all media attachments that are shown in a notification must be stored locally on the device. This means that a push notification can't really have media attachments, unless a service extension is used to download and store the media locally, before appending it to the notification.

Service extensions can only be implemented for push notifications that have the `mutable-content` property added to their `aps` payload, as shown in the following sample:

```
{
  "aps": {
    "alert": "You have a new message!",
    "badge": 1,
    "mutable-content": 1
  },
  "custom-encrypted-message": "MyEncryptedMessage"
}
```

When the `mutable-content` property is detected by iOS, your service extension is activated and receives the notification before it's displayed to the user. A service extension is created in the same way as other extensions. You go to the project settings in Xcode, and in the sidebar that shows all targets, you click the + icon. In the dialog that appears, you select **Notification Service Extension** and give it a name, and Xcode will provide you with the required boilerplate code. When you add a service extension, a sample extension is added to your project to illustrate what a service extension that updates the notification's body text looks like.

Imagine that the **Notifications** app uses a backend server that sends push notifications when a certain reminder is due. To maintain the user's privacy, you send the following payload to the user:

```
{  
    "aps": {  
        "alert": "A reminder is due.",  
        "badge": 1,  
        "mutable-content": 1  
    },  
    "reminder-uuid": "D82ED5AE-C363-46AF-9B0D-301C2E21C58E"  
}
```

The preceding payload contains a very plain message and has the unique identifier for the reminder added as a custom property. You can implement a service extension to read that identifier, and retrieve the appropriate reminder from the Core Data store to show it to the user.

First, add a new **Notification Service Extension** to the Notifications app and name it `ReminderContent`. After doing this, you need to make sure that the extension and the app can both access the Core Data database. Enable the **App Groups** capability for the app and create a new app group identifier. Make sure to enable **App Groups** for the extension as well, and add the extension to the group you just created for the app. Next, make sure the `.xcdatamodeld` and `PersistentHelper.swift` files are added to both the app and the extension target, and then update the code to load the persistent container in `PersistentHelper.swift` as follows:

```
static let persistentContainer: NSPersistentContainer = {  
    let container = NSPersistentContainer(name: "Notifications")  
  
    let containerUrl =  
        FileManager.default.containerURL(forSecurityApplicationGroupIdentifier  
            : "group.com.donnywals.notifications-app")!  
    let databaseUrl =  
        containerUrl.appendingPathComponent("Notifications.sqlite")  
    let description = NSPersistentStoreDescription(url: databaseUrl)
```

```
        container.persistentStoreDescriptions = [description]

        container.loadPersistentStores(completionHandler: {
            (storeDescription, error) in

        })
        return container
    }()
}
```

The process of sharing a Core Data store, as outlined in the preceding section, is described in more detail in [Chapter 20, Streamlining Experiences With Siri](#). Once you have set up the shared Core Data store, let's have a look at the service extension example that Xcode has generated for you.

The boilerplate code that Xcode generates for a service extension is rather interesting. Two properties are created: `contentHandler` and `bestAttemptContent`. The properties are initially given a value in `didReceive(_:withContentHandler:)`. This method is called as soon as the extension is expected to handle the notification.

If the extension fails to call the callback handler in a timely manner, the system calls `serviceExtensionTimeWillExpire()`. This is essentially the last chance to quickly come up with content for the notification. If the extension still fails to call the callback, the pushed notification is displayed to the user in its original form. The default version of `serviceExtensionTimeWillExpire()`, generated by Xcode, simply calls the system callback with the notification immediately to ensure a timely response.

Add the following implementation for `didReceive(_:withContentHandler:)` to retrieve the identifier from the push message payload that was shown previously, and create a new notification with the appropriate reminder. Don't forget to import Core Data at the top of `NotificationService.swift`:

```
override func didReceive(_ request: UNNotificationRequest,
    withContentHandler contentHandler: @escaping (UNNotificationContent)
    -> Void) {
    self.contentHandler = contentHandler
    bestAttemptContent = (request.content.mutableCopy() as?
    UNMutableNotificationContent)

    guard let identifier = request.content.userInfo["reminder-uuid"] as?
    String else {
        contentHandler(request.content)
        return
    }
```

```
let predicate = NSPredicate(format: "identifier == %@", identifier)
let fetchRequest: NSFetchedRequest<Reminder> = Reminder.fetchRequest()
fetchRequest.predicate = predicate

let moc = PersistentHelper.persistentContainer.viewContext
guard let results = try? moc.fetch(fetchRequest),
      let reminder = results.first else {
    contentHandler(request.content)
    return
}

if let bestAttemptContent = bestAttemptContent {
    bestAttemptContent.title = "Reminder"
    bestAttemptContent.body = reminder.title ?? ""
    bestAttemptContent.categoryIdentifier = "reminder"

    contentHandler(bestAttemptContent)
}
}
```

The preceding code extracts data from the pushed notification to find the appropriate reminder to show to the user, and applies this information to the original notification. This means that the server can send a very candid notification that contains no information about the reminder itself, apart from its identifier. It isn't until the notification reaches the device that the notification is transformed and enriched to show the reminder details. Pretty neat, right?

Adding a content extension to your app

The last feature of notifications that you will learn about in this chapter is a feature that can really make your notification experience pop. Content extensions enable developers to take custom notification actions to a whole new level. You already know that 3D-Touching a notification will make custom actions pop up. However, the notification maintains a standard look and feel, which might not be exactly what you want.

Consider receiving an invite for an event. The notification allows you to accept the invite, or decline it. Wouldn't it be great if your calendar popped up inside that notification as well, allowing you to check your calendar without having to actually open your calendar app, before responding to the invite? This is what content extensions are for. When implemented correctly, they can provide users with essential information relevant to the notification that's currently on display.

To demonstrate the possibilities for content extensions, you will use the simple notification you created for the Notifications app, and take it to the next level. Earlier, you scheduled a local notification that simply notified the user about a reminder. The notification itself contained only the notification title and custom actions that allow the user to mark a task as finished.

You will now use a content extension to show more context about the reminder. The content extension will show the user when the notification was due, and whether it is currently pending. In addition to this, you will also update the notification UI when the user taps the **Complete** action, so the UI reflects the current reminder status. If a notification is no longer pending by the time the notification is delivered, the **Complete** action will be replaced with a **Pending** action.

Get started by adding a new extension to your project, and select the **Notification Content Extension** type. Make sure to add your extension to the shared **App Group** you have set up before, and add the Core Data model definition and `PersistentHelper` to the extension target.

When you add a content extension to your project, Xcode generates a small sample implementation for you, which you can use as a reference to see a minimal working implementation for a content extension. Open the `NotificationViewController` class and have a quick look at its content, before removing all generated code and adding the following outlets:

```
@IBOutlet var reminderTitleLabel: UILabel!
@IBOutlet var reminderDueDateLabel: UILabel!
@IBOutlet var reminderStatusLabel: UILabel!
```

Next, add the following implementation for `didReceive(_:)` to set the correct values on the outlets you added:

```
func didReceive(_ notification: UNNotification) {
    guard let identifier =
        notification.request.content.userInfo["reminder-uuid"] as? String
        else { return }

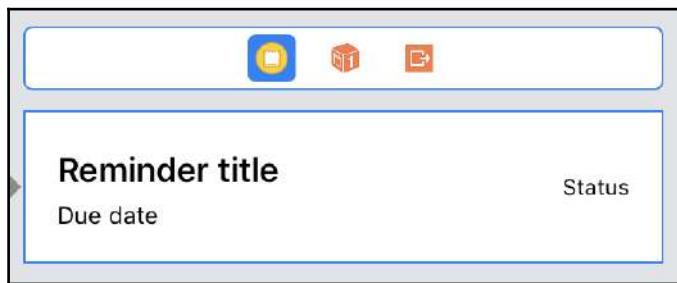
    let predicate = NSPredicate(format: "identifier == %@", identifier)
    let fetchRequest: NSFetchedResultsController<Reminder> = Reminder.fetchRequest()
    fetchRequest.predicate = predicate

    let moc = PersistentHelper.persistentContainer.viewContext
    guard let results = try? moc.fetch(fetchRequest),
        let reminder = results.first
        else { return }
```

```
let dateFormatter = DateFormatter()
dateFormatter.dateStyle = .long
dateFormatter.timeStyle = .medium

reminder.titleLabel.text = reminder.title
reminderDueDateLabel.text = dateFormatter.string(from:
reminder.dueDate!)
reminderStatusLabel.text = reminder.isCompleted ? "Done" : "Pending"
}
```

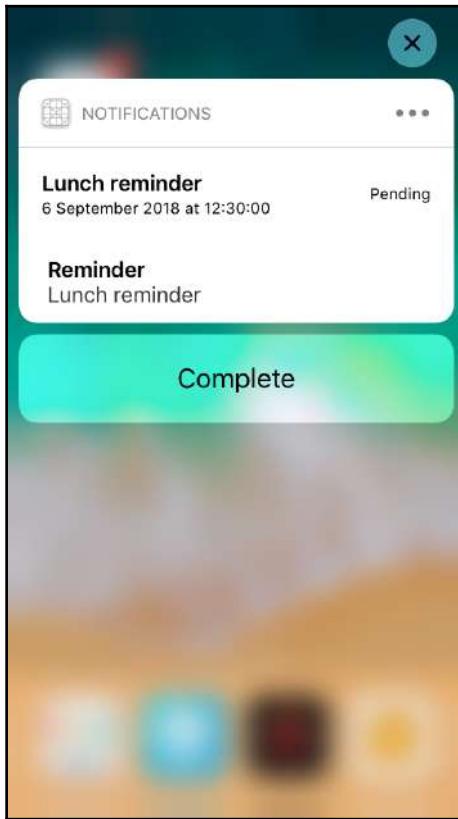
In the storyboard for the content extension, remove the existing label and add three new labels, one for the reminder title, one for the due date, and one for the reminder status. Don't forget to connect the labels to their corresponding outlets, and remove the original outlet that was present in the storyboard. Giving the view controller a height of ±100 points will help you position the labels more easily. You can use stack views and Auto Layout to replicate the following screenshot:



Finally, open the content extension's `Info.plist` file. As mentioned before, content extensions are associated with notifications through categories. The `Info.plist` file is used to specify the category that the current extension should be associated with. Expand the `NSExtension` and `NSExtensionAttributes` properties to find the `UNNotificationExtensionCategory` property. Give this field a value of `reminder`.

You are now ready to take your extension for a spin. Select your extension from the drop-down menu next to the run and stop controls in the top-left corner of Xcode, and run your project. Xcode will ask you for an app to run; pick **Notifications**. Schedule a new reminder in the Notifications app and wait for a notification to appear.

Once the notification appears, the following UI should be shown when you 3D-Touch on the notification:



If you've tested this example yourself, you may have noticed that the notification's custom view was too high initially, and that it animated to the correct size for properly fitting your notification contents. The notification determines its final size using AutoLayout, so the final size won't be known until the notification content is on screen. The animation doesn't look very good, but luckily, there is a way to minimize the amount of resizing that your content extension has to do.

In the extension's `Info.plist` file, there is a property called `UNNotificationExtensionInitialContentSizeRatio`, which sits right below the `notificationCategory` property. The default value for this property is 1, but for your notification, you could probably use a value that's a lot smaller. Try setting this value to 0.2 and run your extension again. It doesn't have to animate nearly as much because the initial height is now just 20% of the extension's width. Much better.

You probably also noticed that the original notification contents are visible below the custom view. You can hide this default content by adding the `UNNotificationExtensionDefaultContentHidden` property to the extension's `Info.plist` file. All the properties you add for your extension should be added at the same level as `UNNotificationExtensionInitialContentSizeRatio` and `UNNotificationExtensionCategory`:

▼ NSExtension	Dictionary	(3 items)
▼ NSExtensionAttributes	Dictionary	(3 items)
UNNotificationExtensionDefault...	Boolean	YES
UNNotificationExtensionCategory	String	reminder
UNNotificationExtensionInitialC...	Number	0.2
NSExtensionMainStoryboard	String	MainInterface
NSExtensionPointIdentifier	String	com.apple.usernotifications.content-extension

This wraps up the interface part of the content extension. In addition to showing a custom UI for a notification, content extensions can respond to actions that the user selects right inside the content extension itself. To do this, you should implement the `didReceive (_:completionHandler:)` delegate method in the extension. Once this method is implemented, the extension becomes responsible for all actions that are chosen by the user. This means that the extension should either handle all the actions, or that it should explicitly pass them on to the host app if the selected action can't be handled within the content extension.

After handling the incoming action, the notification extension determines what should happen next. This is done by calling the completion handler with a `UNNotificationContentExtensionResponseOption`. There are three options to choose from:

- Dismiss the notification and do nothing
- Dismiss the notification and forward the chosen action to the host app
- Keep the extension active so the user can pick more actions or, using Apple's *Messages* app as an example, so that the user can carry on the conversation inside the content extension

Add the following implementation for `didReceive(_:completionHandler:)` to mark a reminder as completed when a user taps the **Complete** action:

```
func didReceive(_ response: UNNotificationResponse, completionHandler: @escaping (UNNotificationContentExtensionResponseOption) -> Void) {

    guard let reminder = extractReminderFromResponse(response) else {
        completion(.dismissAndForwardAction)
        return
    }

    if response.actionIdentifier == "complete-reminder" {
        setCompleted(true, forReminder: reminder, completionHandler: completion)
    } else {
        setCompleted(false, forReminder: reminder, completionHandler: completion)
    }
}

func setCompleted(_ completed: Bool, forReminder reminder: Reminder, completionHandler completion: @escaping (UNNotificationContentExtensionResponseOption) -> Void) {

    reminder.managedObjectContext!.perform {
        reminder.isCompleted = true
        self.reminderStatusLabel.text = reminder.isCompleted ? "Done" : "Pending"

        try! reminder.managedObjectContext!.save()

        completion(.doNotDismiss)
    }
}

func extractReminderFromResponse(_ response: UNNotificationResponse) -> Reminder? {
    let userInfo = response.notification.request.content.userInfo

    guard let identifier = userInfo["reminder-uuid"] as? String else {
        return nil
    }

    let fetchRequest: NSFetchedResultsController<Reminder> = Reminder.fetchRequest()
    fetchRequest.predicate = NSPredicate(format: "identifier == %@", identifier)
```

```
let moc = PersistentHelper.persistentContainer.viewContext
guard let results = try? moc.fetch(fetchRequest),
      let reminder = results.first else {
    return nil
}

return reminder
}
```

The preceding code takes very similar approach to marking the reminder as completed to the one you implemented earlier in `AppDelegate`. Note that the reminder can also be marked as not completed. The reason this is taken into account is that once the user has tapped the **Complete** action, this action should be swapped out for a **Pending** action, in case the user marked the reminder as completed by accident.

Add the following code to replace the action that is shown underneath the content extension:

```
func setNotificationForReminder(_ reminder: Reminder) {
    let identifier = reminder.isCompleted ? "pending-reminder":
"complete-reminder"
    let label = reminder.isCompleted ? "Pending" : "Complete"

    let action = UNNotificationAction(identifier: identifier, title:
label, options: [])
    extensionContext?.notificationActions = [action]
}
```

The preceding method creates and applies a new action, depending on the current reminder status. Add calls to this method after setting the current reminder's completion status and in the initial `didReceive(_:)` method, so that the correct action is shown in the event that the reminder associated with the notification has already been completed.

As you've seen just in the preceding examples, content extensions are not complex to implement, yet they are amazingly powerful. You can add an entirely new layer of interaction and increase the relevance of notifications by implementing a content extension, and it is strongly recommended that you always consider ways to implement extensions for your notifications to provide the user with rich information about their notification as quickly as possible.

Summary

In this chapter, you explored the amazing world of notifications. Even though notifications are simple in nature and short-lived, they are a vital part of the iOS ecosystem, and users rely on them to provide timely, interesting, and relevant updates about subjects that matter to them, right there and then. You saw how to go from the basics, simply scheduling a notification, to more advanced subjects, such as custom actions and implementing grouped notifications. You saw that service extensions allow you to implement great features, such as true end-to-end encryption, or enrich push notifications with content that is stored on the device.

You then implemented a content extension, and saw how a content extension can take a simple, plain notification, and make it interesting, rich, and more relevant. The example of a calendar that appears with event invites comes to mind immediately, but the possibilities are endless. Proper usage of notification extensions will truly make your app stand out in a positive and engaging way.

The next chapter focuses on another extension: the Today extension!

Questions

1. How are `UNUserNotificationCenter` and `NotificationCenter` different?
 - a) One is responsible for showing notifications to users, and the other is a mechanism to send notifications about events inside an app.
 - b) One is supported on iOS 9 and below, while the other is supported on iOS 10 and above.
 - c) They are the same.
2. What kinds of notifications can you send to a user?
 - a) Push notifications and local notifications.
 - b) Push notifications and scheduled notifications.
 - c) Push notifications.

3. What are provisional notifications?

- a) Special notifications that provide critical information to users.
- b) Notifications that are encrypted.
- c) Notifications that are delivered to the user quietly, so they can preview the notifications your app will send.

4. How does iOS group notifications?

- a) It uses the thread identifier from the notification content and the app that the notification belongs to.
- b) By thread identifier only.
- c) By notification category.

5. What does a Notification Service Extension do?

- a) It implements end-to-end encryption.
- b) It allows you to add custom action handlers.
- c) It allows you to replace a notification's contents before the user sees it.

6. What does a notification content extension do?

- a) It allows you to add custom action handlers.
- b) It allows you to create a custom interface for when a user 3D-Touches a notification.
- c) It implements end-to-end encryption.

Further reading

- Apple's Payload Key Reference documentation: https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/generating_a_remote_notification
- Apple's documentation on localizing your app: <https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPInternational/LocalizingYourApp/LocalizingYourApp.html>

23

Instant Information with a Today Extension

When a user swipes right on the **Lock Screen** or in the **Notification Center**, they are taken to section called the **Today View**. In this section, users are presented with several widgets that provide useful information at a glance. These widgets are called **Today Extensions**.

In this chapter, you will learn how you can build a **Today Extension** so you can integrate useful information from your app with the **Today View**. A proper integration with the **Today View** can help users see the information they care about at a glance. The basis for the extension that you will build in this chapter is an inspirational quote app called **The Daily Quote**. In its current state, **The Daily Quote** only shows a quote when the user opens the app. You will expand this app by implementing a **Today Extension**.

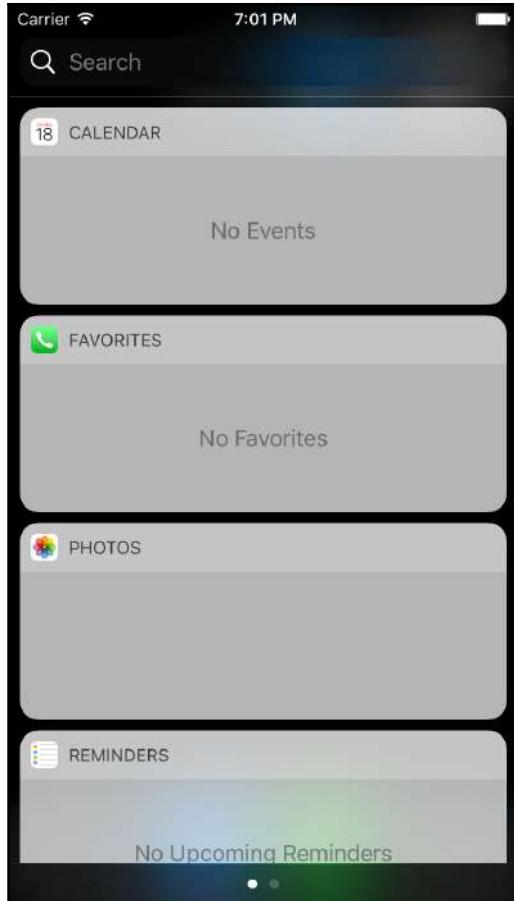
This chapter covers the following topics:

- Understanding the anatomy of a Today Extension
- Adding a Today Extension to your app

Understanding the anatomy of a Today Extension

The terms **Today Extension** and **widget** can be used interchangeably; they both refer to a component that is present in iOS's **Today View**. In this chapter, the term *widget* is used throughout.

If you swipe down from the top of the screen to open **Spotlight** and swipe right after that, you're presented with the **Today View**. On the simulator, this view tends to look rather empty, but on your device, there's probably a lot more going on. The following screenshot shows the **Today View** on the simulator with a couple of widgets added to it:



Users can scroll to the bottom of this view and manage their widgets from there. They can add new widgets and remove existing ones. All these widgets have one thing in common: they provide relevant information for the current moment or day. For the **Calendar**, this means showing events that you have planned for today or tomorrow. The **Favorites** widget in the screenshot usually contains contacts that the user interacts with often. Users can even add a widget that suggests apps that the user is likely to use based on the current time and the user's location.

Any widget that you implement for your app should aim to provide your users with relevant, easy-to-scan information. Doing this makes your widget more useful and more likely to be added to a user's Today View.

Before you implement your own Today Extension, let's take a closer look at how a Today Extension works and where a user can find them in iOS.

Finding Today Extensions in iOS

In iOS, there are several places where people can discover and use widgets. For most scenarios, the way the widget looks will be similar to how it looks in the Today View, apart from some very minor modifications. You are already aware that users can find widgets in the Today View, which can be found by swiping right on the lock screen or Notification Center. In addition to these two places, the Today View is also shown on the leftmost screen on the user's Springboard.

One more place where users can find and use Today Extensions is directly on the home screen. When a user 3D-Touches on an app icon for an app that has implemented a Today Extension, the corresponding widget will pop up, as shown in the following screenshot:



If a user hasn't added the widget for the app that they pressed to the Today View yet, the widget will feature an **Add Widget** button that allows users to quickly add the extension to the Today View so they can access it there in the future.

Understanding Today Extensions

A **Today Extension** isn't much more than a view controller that is embedded inside the notification center's Today View. This means that when your **Today Extension** is initialized, there is no `AppDelegate` involved at all. The extension does not have any application life cycle methods, such as `application(_:didFinishLaunchingWithOptions:)`.

The Today View displays your widget by showing the view controller that you set up for it. This is very similar to how a **Notification Content Extension** works since that extension also has a single view controller to define its functionality.

The Today View is in control of positioning and rendering all of its widgets. This means that you can't set the frame for the widget's view yourself. When you implemented your **Notification Content Extension** in the previous chapter, you could force your notification to be shown at a certain height through AutoLayout. This isn't possible when you implement a **Today Extension**.

Before iOS 10, a widget would resize itself based on its contents or the size that the widget would set as its `preferredContentSize`. Since iOS 10, this behavior has changed, and widgets have a user-defined compact mode and an expanded mode. The compact mode should typically be about 110 points in height, but in reality, this height varies based on the user's text-size preferences.

Because there are no real guarantees regarding the height of your widget can use, it's essential that you only show the most critical information in your widget. Because widgets tend to resize based on the user's text size preferences, it's a good idea to adopt dynamic type in your widget. This makes your widget more accessible by scaling the text inside your widget based on the user's preferences. You'll learn more about this when you implement the widget.

If your widget supports it, the user can switch to the expanded state for your widget. In the expanded state, you have a bit more freedom to set the size of the widget yourself. You will also explore this more when you implement your own widget.

One more key principle of the **Today Extension** is that the widget never communicates with its host app directly. If you have a widget that the user can tap on to open the app, the widget doesn't directly instruct the app to do this. Instead, URLs are used. You can use a custom URL scheme or a Universal Link to achieve this. In several snippets of example code, Apple uses custom URL schemes to redirect users from a widget to an app. **The Daily Quote** app you will extend in this chapter will work similarly.

Now that you're up to speed with **Today Extensions**, it's time to see how you can add your own **Today Extension** to **The Daily Quote**.

Adding a Today Extension to your app

In the code bundle for this chapter, you'll find a project called **The Daily Quote**. It's a straightforward app that displays a different inspirational quote to the user every day. The quotes are hardcoded in the `Quotes.swift` file and the current quote is stored in the `User Defaults` store. The UI contains just two labels to show the current quote and the person that's quoted.

Even though the project is quite simple, each file contains something interesting. If you select one of the labels in the **Storyboard** and examine its attributes, you'll find that the font for the quote itself is **Title 1** and the font for the quoted person is **Caption 1**. This is different from the default system font that you usually use. When you configure labels with these predefined styles, the text in the label will dynamically adjust based on the user's preferences. Adopting dynamic type such as this is requires only minimal effort and is great from an accessibility standpoint.



If your interface allows it, it's recommended that you make use of accessible type, simply because it will make your app more comfortable to use for all of your users.

If you look at the `QuoteViewController` class, there are only a couple of lines involved in displaying a quote. The reason this code is so simple and concise is that a great deal of preparatory work was done in the `Quote` model file. Go ahead and open that file to see what's going on.

The `Quotes` struct is set up in such a way that it can be used as a simple database. The struct contains several static properties and methods to provide quotes from a predetermined list of quotes.

If you were to build a similar app and put it in the App Store, you'd probably want to download the quotes from a server and store it in Core Data, because pushing an update every time you want to add or remove a couple of quotes is a lot of effort for a simple change. There are only a couple of constant instance properties present on the `Quote` struct:

```
let text: String  
let creator: String
```

These properties are the ones that `QuoteViewController` reads and displays to the user.

Furthermore, `User Defaults` is used to store which quote should be shown for the current day. `User Defaults` is a simple data store that stores app-related settings. It's essentially a lightweight persistence layer that you can use to store simple objects.

The Daily Quote makes use of `User Defaults` to store the date on which the current quote was set as well as the index in the list of `Quote` instances that points to the current quote:

```
static var current: Quote {  
    if let lastQuoteDate = userDefaults.object(forKey: "lastQuoteDate")  
        as? Date {  
        if NSCalendar.current.compare(Date(), to: lastQuoteDate,  
            toGranularity: .day) == .orderedDescending {  
            setNewQuote()  
        }  
    } else {  
        setNewQuote()  
    }  
  
    guard let quoteIndex = userDefaults.object(forKey: "quoteIndex") as?  
        Int,  
        let quote = Quote.quote(atIndex: quoteIndex)  
        else { fatalError("Could not create a quote..") }  
  
    return quote  
}  
  
static func setNewQuote() {  
    let quoteIndex = Quote.randomIndex  
    let date = Date()  
  
    userDefaults.set(date, forKey: "lastQuoteDate")  
    userDefaults.set(quoteIndex, forKey: "quoteIndex")  
}
```

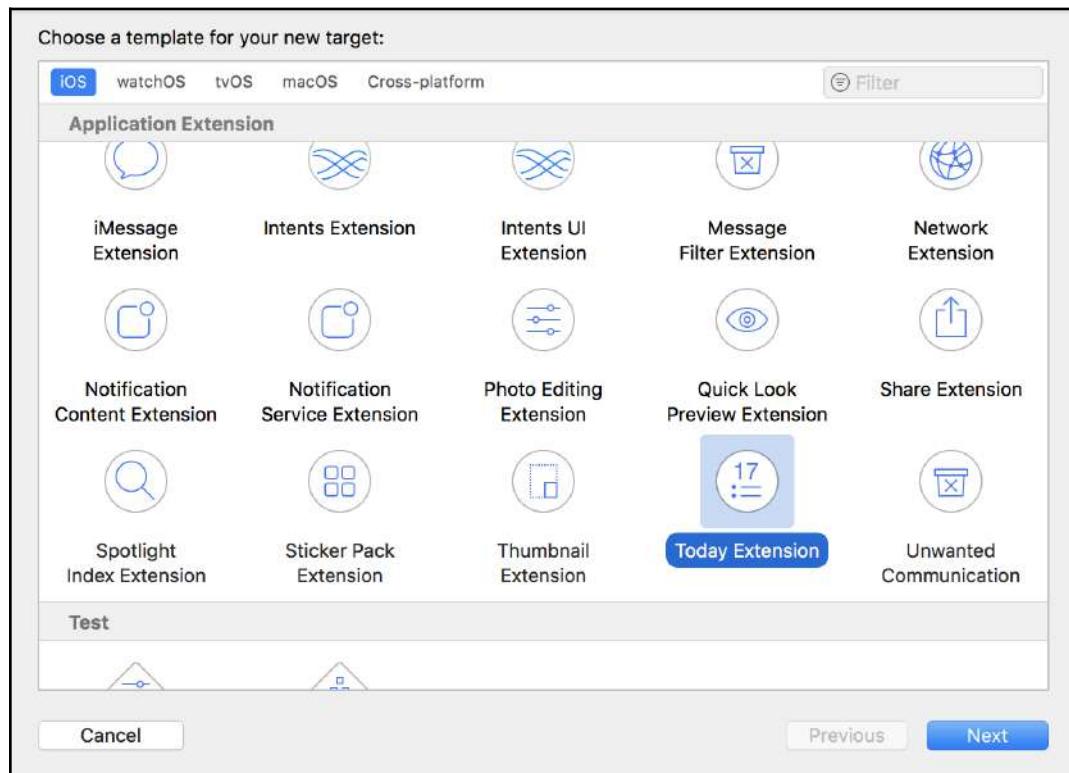
The preceding snippet illustrates the process of retrieving and storing the current quote. First, the code checks whether a quote has been set before and if so, it makes sure that the quote is at least a day old before generating a new one. Next, the current quoteIndex is retrieved from UserDefaults and returned as the current quote.

If you need a quick refresher on how to use UserDefaults in your app, go back to Chapter 15, *Syncing Data with CloudKit*.



Don't use UserDefaults to store privacy-sensitive data. It's not considered secure; the keychain should be used for this purpose. Also, make sure that you're not storing complex or repetitive data in UserDefaults. It's not a database, nor is it optimized for reading and writing many times. Stick to simple, application-specific settings only.

Now that you are familiar with the contents of the starter project, add a new target to your project just like you have done before, but this time, select the **Today Extension**:



To see the default widget that Xcode generates for you, go ahead and run your extension. The first time you do this, your widget might not show up immediately. Don't worry if this is the case, build and run again, and your app should show up as a widget in the Today View.

Now let's see what kind of files and boilerplate code Xcode has generated for you. In the folder that Xcode created for your extension, you'll find a view controller, a storyboard and an `Info.plist` file. This structure is very similar to the structure that was used for the **Notification Content Extension** you created in the previous chapter.



You can change the size of the view controller in your Storyboard if you like, but note that the display size for the widget is taller than the visible size in the storyboard. This is due to the standard compact widget height that iOS uses for your extension. You can't influence this, so you'll need to keep in mind that you only have limited space for your extension, so you should make sure your layout is flexible enough to look good at the compact size the Today View imposes on the widget.

To give the labels in the extension a little bit of room to breathe, select the view controller (not the view itself) and click on the **Size** inspector. Here you'll see that the simulated size for the view controller is set to free-form and you can set a custom width and height. Leave the width as it is for now and set the height to 110. This size should be the smallest size at which our widget displays, and it gives you plenty of room to create the interface. Also, delete the default label that has been added to the interface automatically.

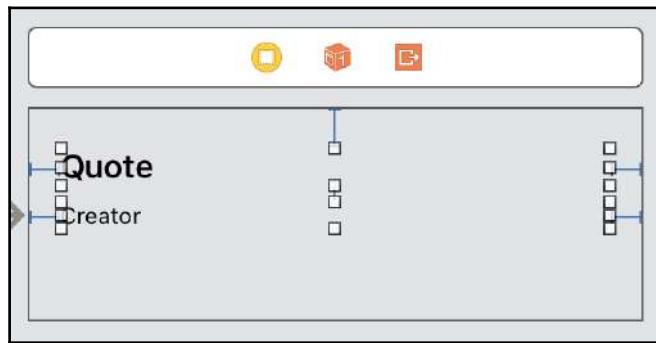
Drag a `UILabel` into the view and set its font to **Headline**. Click the **Font** icon in the **Attributes** inspector to change the font and select the **Font** dropdown to find the dynamic text styles. Position the label in the top-left corner using the blue helper lines and adjust its width, so it covers the available space. After doing this, add the following constraints to the label:

- Leading space to Safe Area
- Top space to Safe Area
- Trailing space to Safe Area

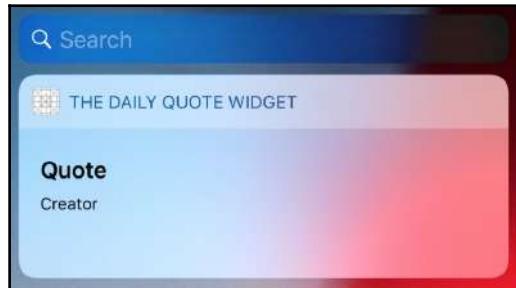
Finally, set the number of lines for the label to 3 so the quote doesn't take up more than 3 lines. Now, drag out another label and position it right below the first label. Make sure this label has the same spacing from the superview's left edge and that it has the same width as the other label. Set its font style to **Caption 1**. Also, add the following constraints to this view:

- Leading space to Safe Area
- Trailing space to Safe Area
- Vertical spacing between this label and the label above it

Your layout in **Interface Builder** should resemble the following screenshot:



Go ahead and rerun your extension. Your final result should look similar to the one shown in the following screenshot:



Now that you have set up the layout for your extension, let's some outlets to the widget view controller so you can display the quote of the day inside of the widget:

```
@IBOutlet var quoteLabel: UILabel!
@IBOutlet var quoteCreator: UILabel!
```

Open the widget's Storyboard file and connect the outlets as you've done before. Select the view controller, go to the **Outlet Inspector**, and drag from the **Outlet Inspector** to the corresponding views to connect the outlets to the view.

The next step to build this widget is to load and display the quote of the day in the widget. Since extensions do not communicate with their host apps directly, you will have to use App Groups again to implement a shared `UserDefault`s store. Sharing `UserDefault`s through App Groups will allow both your widget and app to retrieve the current quote index from `UserDefault`s, and ensures that both interfaces always show the same quote.

Create a new App Group for **The Daily Quote** and make sure the app and extension both belong to the same App Group. Also, make sure that you add `Quote.swift` to the extension target as well as the app target.



If you need a quick refresher on what App Groups are and how they work, go back to Chapter 20, *Streamlining Experiences with Siri* for more information.

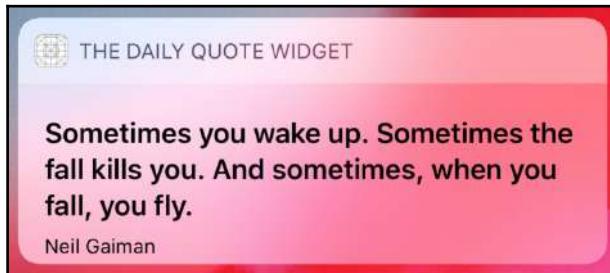
After enabling App Groups and making sure the app and the extension share the `Quote` struct, add the following code to `TodayViewController.swift`:

```
override func viewDidLoad() {
    super.viewDidLoad()

    updateWidget()
}

func updateWidget() {
    let quote = Quote.current
    quoteLabel.text = quote.text
    quoteCreator.text = quote.creator
}
```

If you run the app now, you should see the current quote for the day shown in your widget, as illustrated in the following screenshot:



The widget currently loads the quote when its view controller is loaded. There are no guarantees about how and when your widget will be recreated. This means that you can't rely on `viewDidLoad()` to make sure that your widget always contains the most recent information. In the example that Xcode generated when you first created your extension, a method called `widgetPerformUpdate(completionHandler:)` can be found. This method is called regularly by the system to make sure that your widget can update its contents when needed.

The system passes a `completionHandler` to the update method that you must call to let iOS know whether your widget has loaded new data. This allows iOS to optimize the frequency and timing of calls to

`widgetPerformUpdate(completionHandler:)` so your widget isn't doing more work than needed, and to make sure it updates when it's most likely to have new data. Add the following implementation for this

`widgetPerformUpdate(completionHandler:)` to `TodayViewController.swift`:

```
func widgetPerformUpdate(completionHandler: @escaping (NCUpdateResult) -> Void) {
    let currentText = quoteLabel.text
    updateWidget()
    let newText = quoteLabel.text

    if currentText == newText {
        completionHandler(NCUpdateResult.noData)
    } else {
        completionHandler(NCUpdateResult.newData)
    }
}
```

The preceding method updates the widget by calling `updateWidget()`, just like `viewDidLoad` does. Before doing this, the current text for the quote is stored. After updating the widget, the new text for the quote is stored. Based on the comparison of these two strings, the callback is informed about the result of the update request.

Some widgets in the Today View have a **Show More** button that can be tapped to expand the widget. You can implement this button in your own widget by setting the `widgetLargestAvailableDisplayMode` property on the notification context to the `.expanded` value. Setting the largest display mode to expanded tells the Today View that your widget needs the ability to show contents in a larger view. The following line of code demonstrates how to set the largest available display mode:

```
extensionContext?.widgetLargestAvailableDisplayMode = .expanded
```

When the user taps the **Show More** button that is rendered after setting the largest display mode, the `widgetActiveDisplayModeDidChange(_:withMaximumSize:)` method is called on your notification extension. In this method, you can set your `preferredContentSize` of `TodayViewController` to the size you would like your widget to be. This should always be smaller than the maximum size that is passed to the method because the Today View will not make your widget any bigger than the maximum size.

The following code is a short sample implementation for `widgetActiveDisplayModeDidChange(_:withMaximumSize:)` that determines the preferred content size based on the new display mode:

```
func widgetActiveDisplayModeDidChange(_ activeDisplayStyle:  
NCWidgetDisplayStyle, withMaximumSize maxSize: CGSize) {  
  
    if activeDisplayStyle == .compact {  
        preferredContentSize = maxSize  
    } else {  
        preferredContentSize = CGSize(width: maxSize.width, height: 200)  
    }  
}
```

With this information, you know everything you need to know to start implementing widgets for your own applications.

Summary

This chapter has shown you everything you need to know about implementing a simple Today Extension that shows a new inspirational quote to your users every day. You learned where users can find and use widgets on iOS and how they work. In addition, you learned how to use App Groups to share a `UserDefault`s store between extensions and apps, which is similar to how you can share a Core Data store in an App Group.



As an exercise, try to change how often a new quote is shown to the user. For instance, try loading a new quote every couple of hours to keep the user entertained.

You also learned how to make sure your widget always has up-to-date data and how you can allow your user to expand a widget, so it can show even more information. As a side note, you also learned that you can adopt accessible and resizable text by using Apple's predefined text styles to improve your widget's accessibility.

In the next chapter, you will learn how to improve an app that you have worked on before by implementing Drag and Drop features for it!

Questions

1. Where can users find Today Extensions?
 - a) By Swiping right on the lock screen.
 - b) By Swiping right on the lock screen or Notification Center, and on the leftmost screen on the Springboard.
 - c) By Swiping right on the lock screen or Notification Center, on the leftmost screen on the Springboard, and 3D-touching an app.
2. At what height are widgets displayed by default?
 - a) It depends on the screen width and user settings.
 - b) 100 points.
 - c) It depends on the size of your contents.

3. How can you allow a user to expand your widget's height?
 - a) By setting the preferred content size.
 - b) By implementing `widgetActiveDisplayModeDidChange(_:withMaximumSize:)`.
 - c) By setting the largest available display mode on the extension context to expanded.
4. What is the best place to update your widget contents?
 - a) In `viewDidLoad()`.
 - b) In `widgetPerformUpdate(completionHandler:)`.
 - c) In `viewDidAppear()`.
5. How does iOS know the best moment to reload a widget's contents?
 - a) It reloads when the widget is about to appear.
 - b) It monitors network requests.
 - c) With the completion handler that is passed to `widgetPerformUpdate(completionHandler:)`.

Further reading

- Accessibility Programming Guide: <https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/iPhoneAccessibility/Introduction/Introduction.html>
- Today Extension Programming Guide: <https://developer.apple.com/library/archive/documentation/General/Conceptual/ExtensibilityPG/Today.html>

24

Exchanging Data With Drag And Drop

Until iOS 11 came out, iOS had a pretty terrible drag and drop experience. If you wanted to move a couple of photos from one app on your iPad to another, you would usually have to copy and paste each photo individually. If you compare this experience with the desktop, where you can grab one or multiple items and drag those items from one place to the other, the iOS experience sounds pretty bad.

Since iOS 11, users can finally enjoy a full drag and drop experience on iOS. Implementing drag and drop in your own apps has been made surprisingly simple since all you need to do is implement a couple of required methods from a handful of protocols. This chapter's purpose is to show you how to handle drag and drop in the augmented reality gallery you built in Chapter 16, *Using Augmented Reality*. The gallery app will be expanded so that users can add photos from their own photo library, the internet, or any other source to their art library.

This chapter covers the following topics:

- Understanding the drag and drop experience
- Implementing a basic drag and drop functionality
- Customizing the drag and drop experience

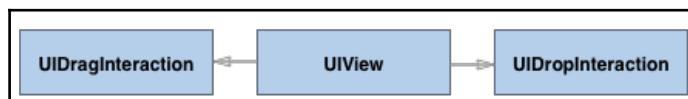
Understanding the drag and drop experience

The drag and drop experience on iOS is quite simple to use; pick up an item on the screen, drag it somewhere else, and let it go to make the dragged item appear in the new place. However, iOS didn't have this behavior until iOS 11. And even now, its full range of capabilities is only available on the iPad. Despite this limitation, the drag and drop experience is really powerful on both the iPhone and iPad.

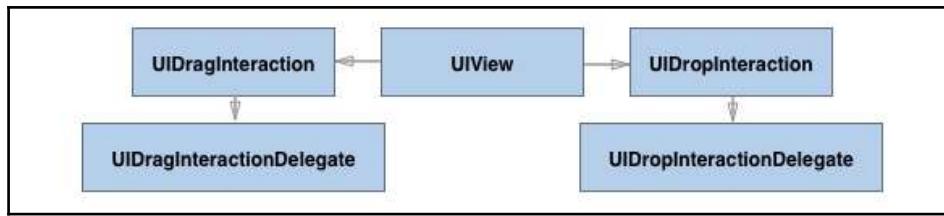
Users can pick up an item from any app, and move it over to any other app that implements drag and drop, as they please. And dragging items is not even limited to just a single item, it's possible for users to pick up multiple items in a single drag session. The items that a user adds to their drag session don't have to be of the same type; this makes the drag and drop experience extremely flexible and fluid. Imagine selecting some text and a picture in Safari and dragging them over to a note you're making. Both the image and the text can be added to a note in just a single gesture.

Unfortunately, apps are not able to handle drag and drop out of the box; you'll need to do a little bit of work to support this feature. At the heart of the drag and drop experience are two protocols and two new interaction classes. Let's briefly go over the protocols and classes before going more in-depth and implementing drag and drop for your augmented reality art gallery.

The first requirement for a drag and drop session is a view that can start or receive a drag session. This ability is added to a view through either `UIDropInteraction` or `UIDragInteraction`. Both interactions are subclasses of the `UIInteraction` base class. `UIInteraction` manifests itself similarly to `UIGestureRecognizer` in the sense that you create an instance of it and attach it to a view. The following figure shows the relationship between a view and the drag and drop interactions:



When adding either `UIDragInteraction` or `UIDropInteraction` to `UIView`, you must also set up a delegate for the interaction you're adding. When you add `UIDropInteraction`, you must set `UIDropInteractionDelegate` on it. When you add `UIDragInteraction`, you must assign `UIDragInteractionDelegate` for it. The following figure illustrates the relationship between the interactions and their delegates:



Each protocol is related to a different set of responsibilities, as is suggested by their names. Let's have a look at `UIDragInteractionDelegate` first.

Understanding `UIDragInteractionDelegate`

Any time the user long-presses an item that has `UIDragInteraction` associated with it, a new drag session is started. This drag session uses the interaction's `UIDragInteractionDelegate` to obtain the data for the item that is being dragged, a drag preview, and more.

`UIDragInteractionDelegate` only has a single required method: `dragInteraction(_:itemsForBeginning:)`. If you implement this method, your app can start supplying data to other apps. The return type for `dragInteraction(_:itemsForBeginning:)` is `[UIDragItem]`. `UIDragItem` is a container for `NSItemProvider`. The drag item is also responsible for providing a preview of the dragged item while it's being dragged. Normally, the selected view will be used, but sometimes you might want to provide a custom view for the preview. If this is the case, you can provide your own preview through `UIDragItem`.

A straightforward example of `dragInteraction(_:itemsForBeginning:)` looks as follows:

```
func dragInteraction(_ interaction: UIDragInteraction,
itemsForBeginning session: UIDragSession) -> [UIDragItem] {
    let text = "Hello, world"
    let provider = NSItemProvider(object: text as NSString)
    let item = UIDragItem(itemProvider: provider)
    return [item]
}
```

As mentioned before, users can add multiple objects to an existing drag session by tapping on them. Whenever the user taps a draggable item while a session is in progress, `dragInteraction(_:itemsForAddingTo:withTouchAt:)` is called on `UIDragInteractionDelegate`.

If you return an empty array in this method, the tapped item will not be added to the drag session. But if you return one or more `UIDragItem` instances, those items will be appended to the existing drag session.

A drag session can be divided into three stages:

- **The lift stage:** This is the stage where the first drag item is requested, and the view that is about to be dragged is animated, so the user sees that they have started a drag session.
- **The dragging stage:** The user is now moving the drag item around.
- **The end stage:** Either a drop has been performed, or the drag session got canceled. If a session is canceled, the dragged item is animated back to its starting position.

You can monitor and respond to each of the stages through `UIDragInteractionDelegate`. To customize the lift animation, you could implement `dragInteraction(_:willAnimateLiftWith:session:)`, for instance. For a full overview of available drag customizations, you should take a look at the documentation for `UIDragInteractionDelegate`; there are a bunch of methods available for you to implement!

Now that you know how to set up your app for dragging, let's see how dropping works.

Understanding `UIDropInteractionDelegate`

Similar to how `UIDragInteractionDelegate` works, `UIDropInteractionDelegate` is used to respond to different stages in the drag and drop life cycle. Even though `UIDropInteractionDelegate` has no required methods, there are at least two methods that you should implement to support drag and drop. The first method is `dropInteraction(_:sessionDidUpdate:)`.

As soon as the user moves their finger on top of a drop target, the drop target is asked whether it can handle a drop with the contents from the current drop session. To do this, `dropInteraction(_:canHandle:)` is called on the drop target. Assuming the data can be handled, `dropInteraction(_:sessionDidUpdate:)` is called next. You must return an instance of `UIDropProposal` from this method. A drop proposal lets the session know what you'd like to happen if the drop is executed at some point. For instance, you can make a copy proposal or a move proposal. The UI surrounding the contents that are being dragged will update accordingly to let the user know what will happen if they perform the drop.

Now let's say your app can only handle objects of a particular type. You should implement `dropInteraction(_:canHandle:)`. You can use this method to inspect whether the drop session contains items that are relevant for your app to handle. An example of this looks as follows:

```
func dropInteraction(_ interaction: UIDropInteraction, canHandle
session: UIDropSession) -> Bool {
    for item in session.items {
        if item.itemProvider.canLoadObject(ofClass: UIImage.self) {
            return true
        }
    }
    return false
}
```

This example searches for at least one image in the current drop session. You can use `canLoadObject(ofClass:)` on `NSItemProvider` to figure out whether the item provider contains an instance of a particular class, in this case, `UIImage`. If your app restricts drag and drop to the same application, you might not need to implement this method because you know exactly what items in your app are draggable. It is recommended that you always make sure the session can be handled by the drop target.

The second method you should always implement is `dropInteraction(_:performDrop:)`. If you don't implement this method, your app doesn't have a proper implementation of the drop interaction. Once the user drops an item onto a drop target, they expect something to happen. `dropInteraction(_:performDrop:)` is the perfect place to do so.

An example implementation of `dropInteraction(_:performDrop:)` could look as follows:

```
func dropInteraction(_ interaction: UIDropInteraction, performDrop
session: UIDropSession) {
    for item in session.items {
        if item.itemProvider.canLoadObject(ofClass: UIImage.self) {
            item.itemProvider.loadObject(ofClass: UIImage.self) { [weak
self] item, error in
                // handle the item
            }
        }
    }
}
```

The preceding code loops through all items in the drop session. If an item can provide an image, the image is loaded. Note that `loadObject (ofClass:)` uses a callback. This means that the data is loaded asynchronously. The dropped data could be huge and, if it were loaded and processed on the main thread by default, it would make your app freeze for a while. Because Apple made `loadObject (ofClass:)` asynchronous, your app's responsiveness is guaranteed, and your users won't notice any freezes or lag.

Just like a drag session, a drop session typically has three stages:

- The session starts when a user drags content onto a drop target.
- Performing the drop; this is when the user lifts their finger while it's on a drop target.
- Ending the drop; the user has either dragged their finger away from the drop target, or the drop has been performed successfully.

Apple has not defined the stages of the drag and drop experience as you have just learned them. Dividing the experience into the preceding steps is merely intended to help you to grasp the life cycle of drag and drop. If you want to learn everything about the drag and drop life cycle, make sure to check out the documentation for both `UIDropInteractionDelegate` and `UIDragInteractionDelegate`.

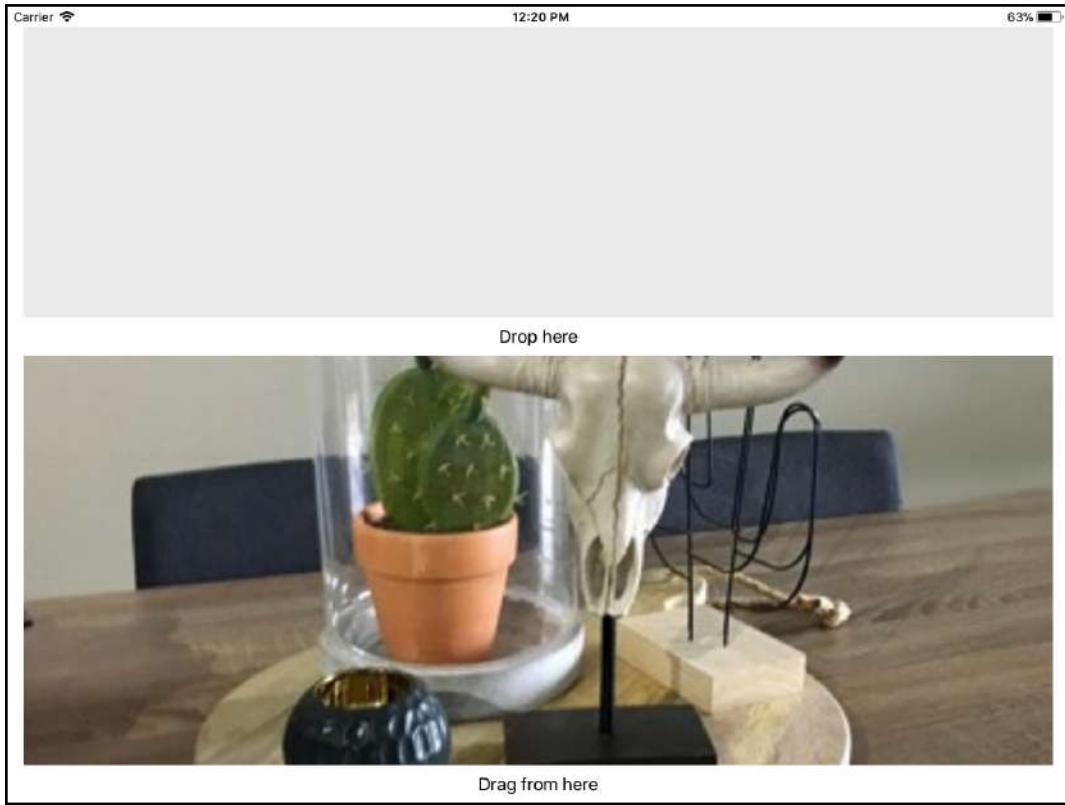
Alright, now that you know what drag and drop looks like in theory, let's see what it looks like in practice!

Implementing a basic drag and drop functionality

The previous section explained how drag and drop works from a theoretical point of view. This section focuses on implementing drag and drop in a sample app. First, you will learn how a simple, regular implementation of drag and drop might work. Next, you'll see how Apple has implemented drag and drop for `UICollectionView` and `UITableView`. These two components have received special treatment for drag and drop, making it even easier to implement drag and drop in apps that use these components.

Adding drag and drop to a plain UIView

Before you implement drag and drop in the Augmented Reality gallery, let's see how you can implement a simple version of drag and drop with a simple view and an image. In the code bundle for this chapter, you'll find a sample project named **PlainDragDrop**. Open the starting version for this project and run it on an iPad simulator. You should see the user interface shown in the following screenshot:



The goal of this example is to allow users to perform the following actions:

- Drag the image to the drop area.
- Drag an external image to the drop area.
- Drag the bottom image to an external app.

While this might sound like you are going to have to do a lot of work, it's quite simple to implement all three features at once. Simply implementing the first feature enables the other two! Quite convenient, right? You are going to implement drag and drop for this example in just three simple steps:

1. Make `ViewController` conform to `UIDragInteractionDelegate`.
2. Make `ViewController` conform to `UIDropInteractionDelegate`.
3. Add interactions to the image and drop area.

If you feel confident that the theoretical explanation from the previous section left you with enough knowledge to implement these steps on your own, that's great! You should give implementing this on your own a go and then refer back to the following code snippets if you get stuck or to check the work you've done. If you prefer to follow along instead, that's great too.

The first step to enable drag and drop in the sample app is to make `DragDropViewController` conform to `UIDragInteractionDelegate`. To do this, you only have to implement a single method. As shown in the explanation about the drag and drop experience, the only required method on `UIDragInteractionDelegate` is `dragInteraction(_:itemsForBeginning:)`. This method requires you to provide a `UIDragItem` for the drag session. Since this example will not support dragging multiple items from the app, there is no need to implement other delegate methods.

Add the following extension to `DragDropViewController.swift` to implement `UIDragInteractionDelegate`:

```
extension DragDropViewController: UIDragInteractionDelegate {  
    func dragInteraction(_ interaction: UIDragInteraction,  
    itemsForBeginning session: UIDragSession) -> [UIDragItem] {  
        guard let image = imageView.image  
        else { return [] }  
  
        let itemProvider = NSItemProvider(object: image)  
        return [UIDragItem(itemProvider: itemProvider)]  
    }  
}
```

The preceding code should look familiar to you since it closely resembles the example code from the previous section. Now that `DragDropViewController` conforms to `UIDragInteractionDelegate`, let's make it conform to `UIDropInteractionDelegate` as well. Add the following extension to `DragDropViewController.swift`:

```
extension DragDropViewController: UIDropInteractionDelegate {
    func dropInteraction(_ interaction: UIDropInteraction,
sessionDidUpdate session: UIDropSession) -> UIDropProposal {
        return UIDropProposal(operation: .copy)
    }

    func dropInteraction(_ interaction: UIDropInteraction, performDrop
session: UIDropSession) {
        guard let itemProvider = session.items.first?.itemProvider,
            itemProvider.canLoadObject(ofClass: UIImage.self)
        else { return }

        itemProvider.loadObject(ofClass: UIImage.self) { [weak self]
            loadedItem, error in
            guard let image = loadedItem as? UIImage
            else { return }

            DispatchQueue.main.async {
                self?.dropArea.image = image
            }
        }
    }
}
```

Again, this implementation should look familiar to you. The first method, `dropInteraction(_:sessionDidUpdate:)`, returns a copy proposal since we don't want to move data around. The second method is `dropInteraction(_:performDrop:)`. This method retrieves the image from `itemProvider` that has been created in `UIDragInteractionDelegate` and sets the loaded image as the image for `dropArea`.

The next step is to add the drag and drop interaction objects to the correct views, as follows:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let dragInteraction = UIDragInteraction(delegate: self)
    imageView.addInteraction(dragInteraction)

    let dropInteraction = UIDropInteraction(delegate: self)
    dropArea.addInteraction(dropInteraction)
}
```

Now that all interactions are set up, go ahead and run the app on an iPad. You'll be able to drag the bottom image to the top section, and the image will appear in the top area. If you run the Photos app alongside **PlainDragDrop**, you can drag the bottom image to the photos app, and it will be added to photos. If you drag an image from the Photos app to the top section, the image from Photos will be set as the image for the drop area. Pretty cool stuff! And it was pretty simple to cover all these cases.

Even though the iPhone does not have full drag and drop support, you can still allow a user to drag the image from the top area to the bottom area. Since the iPhone does not support a full drag and drop experience, all drag and drop interaction objects you create are disabled by default. If you want to allow users to use drag and drop in your iPhone app, you must explicitly enable the interaction objects you would like to support. Add the following code to `viewDidLoad` in `DragDropViewController` to enable the drag interaction in the **PlainDragDrop** app:

```
dragInteraction.isEnabled = true
```

Go ahead and test the app on an iPhone; you should be able to drag the top image around. Now that you know how to enable drag and drop for a plain view, let's see how you can add it to a collection view.

Adding drag and drop to a UICollectionView

A lot of iOS apps make extensive use of collections and tables. Therefore, it makes a lot of sense that, whenever Apple introduces a huge feature such as drag and drop, they take a step back and evaluate how the feature should work for collections or tables. Luckily, drag and drop was no exception and Apple truly put some thought into making drag and drop work great.

In this section, you'll implement drag and drop for the collection of images that is at the bottom of the screen for the **ARGallery** app you created in *Chapter 16, Using Augmented Reality*. You will implement the following features that use drag and drop for `UICollectionView`:

- Dragging photos from the collection into the AR viewport.
- Reordering items in the collection view.
- Adding items from external sources to the collection view.

As a bonus, you will make sure that the first two features also work on an iPhone since these features are only used inside the app so they can be used on iPhone. Since Apple has tailored drag and drop to work perfectly with UICollectionView, the basic concepts for drag and drop still apply; you only have to use slightly different protocols. For instance, instead of implementing UIDragInteractionDelegate, you implement UICollectionViewDragDelegate.

The first feature, dragging photos from the collection of images to the AR Gallery, is implemented similarly to the drag and drop experience you implemented before. You will implement the relevant protocols first, and then you will implement the interactions. The code bundle for this chapter contains a slightly modified version of the **ARGallery** that you built before. The modifications allow you to focus on implementing drag and drop instead of having to make minor adjustments to the existing code.

Since you should be familiar with the dropping implementation already, add the following extension to ViewController.swift in AugmentedRealityGallery:

```
extension ViewController: UIDropInteractionDelegate {
    func dropInteraction(_ interaction: UIDropInteraction,
    sessionDidUpdate session: UIDropSession) -> UIDropProposal {
        return UIDropProposal(operation: .copy)
    }

    func dropInteraction(_ interaction: UIDropInteraction, performDrop
    session: UIDropSession) {
        guard let itemProvider = session.items.first?.itemProvider,
              itemProvider.canLoadObject(ofClass: UIImage.self)
        else { return }

        itemProvider.loadObject(ofClass: UIImage.self) { [weak self] item
        ,error in
            guard let image = item as? UIImage
            else { return }

            DispatchQueue.main.async {
                self?.addImageToGallery(image)
            }
        }
    }
}
```

Nothing crazy happens in this snippet. In fact, it's so similar to the code you have seen already that you should be able to understand what happens in this snippet on your own. The next step is to implement the `UICollectionViewDragDelegate` protocol. Add the following extension to `ViewController.swift`:

```
extension ViewController: UICollectionViewDragDelegate {
    func collectionView(_ collectionView: UICollectionView,
itemsForBeginning session: UIDragSession, at indexPath: IndexPath) ->
[UIDragItem] {
    let image = UIImage(named: images[indexPath.row])!
    let itemProvider = NSItemProvider(object: image)

    return [UIDragItem(itemProvider: itemProvider)]
}
}
```

The preceding implementation serves the same purpose as `UIDragInteractionDelegate`. The main difference is that you have access to the `IndexPath` of the item that was selected for dragging. You can use `IndexPath` to obtain the dragged image and create a `UIDragItem` for it. Let's set up the interaction objects for this part of the app now. Add the following lines of code to `viewDidLoad()`:

```
collectionView.dragDelegate = self
collectionView.dragInteractionEnabled = true

let dropInteraction = UIDropInteraction(delegate: self)
arKitScene.addInteraction(dropInteraction)
```

If you build and run the app now, you can drag photos into the AR gallery from the collection view at the bottom. If you run the app on an iPad, you are even able to drag images from external apps into the AR Gallery! This is quite awesome, but you're not done yet. Let's allow users to drag items from external apps into the collection, so they have easy access to it. And while we're at it, let's implement the reordering of the collection using drag and drop as well.

To implement the reordering of items in the collection view and to allow users to add external images to their collection, you must implement `UICollectionViewDropDelegate`. As you will see soon, it is possible to distinguish between a drop session that originated from within the app or outside the app. This information can be used to determine whether the user wants to reorder the collection or add an item to it. Add the following extension for `ViewController`:

```
extension ViewController: UICollectionViewDropDelegate {
    func collectionView(_ collectionView: UICollectionView,
dropSessionDidUpdate session: UIDropSession, withDestinationIndexPath
destinationIndexPath: IndexPath?) -> UICollectionViewDropProposal {

        if session.localDragSession != nil {
            return UICollectionViewDropProposal(operation: .move, intent:
.insertAtDestinationIndexPath)
        }

        return UICollectionViewDropProposal(operation: .copy, intent:
.insertAtDestinationIndexPath)
    }
}
```

The preceding snippet implements

`collectionView(_:dropSessionDidUpdate:withDestinationIndexPath:)`.

This delegate method is pretty similar to

`dropInteraction(_:sessionDidUpdate:)` except you also have access to the destination index path that the user is currently hovering their finger over. By checking whether there is a `localDragSession` on a `UIDropSession`, you can detect whether the user wants to reorder the collection or whether they are adding an item from an external source. By specifying an intent on the drop proposal, `CollectionView` knows how it should update its interface to visualize the action that is taken when the user performs the drop. Speaking of performing drops, add the following code to the `UICollectionViewDropDelegate` extension you just added:

```
unc collectionView(_ collectionView: UICollectionView, performDropWith
coordinator: UICollectionViewDropCoordinator) {
    switch coordinator.proposal.operation {
    case .copy:
        performCopy(forCollectionView: collectionView, with: coordinator)
    case .move:
        performMove(forCollectionView: collectionView, with: coordinator)
    default:
        return
    }
}
```

This method is relatively simple. Depending on the proposal that is used, a different method is called. You will add the copy and move methods soon, but first, let's talk a little bit about `UICollectionViewDropCoordinator`.

`UICollectionViewDropCoordinator` contains information about the items that are being dragged, the animations that should be performed, the drop proposal, and of course the drop session. When performing a drop, you make use of the coordinator to request the drag items, but also to make sure the collection view properly updates its view.

The first method you will implement is

`performMove(forCollectionView:with:)` since it's the simpler of the two remaining methods to implement. Add the following snippet to the `UICollectionViewDropDelegate` extension:

```
func performMove(forCollectionView collectionView: UICollectionView,
with coordinator: UICollectionViewDropCoordinator) {
    let destinationIndexPath = coordinator.destinationIndexPath ?? IndexPath(item: 0, section: 0)

    guard let item = coordinator.items.first,
          let sourceIndexPath = item.sourceIndexPath
    else { return }

    let image = images.remove(at: sourceIndexPath.row)
    images.insert(image, at: destinationIndexPath.row)

    collectionView.performBatchUpdates({
        collectionView.deleteItems(at: [sourceIndexPath])
        collectionView.insertItems(at: [destinationIndexPath])
    })

    coordinator.drop(item.dragItem, toItemAt: destinationIndexPath)
}
```

The preceding snippet uses the coordinator to retrieve the first item in the drag session. The app doesn't support dragging multiple items at once, so this is alright. Next, the item's source index path is used to remove the image that should be moved from the array of images. The destination index path is then used to add the image back into the array of images at its new location. This is done to make sure the data source is updated before updating the collection view. After the collection view is updated, `drop(_:toItemAt:)` is called on the coordinator to animate the drop action.

The final method you need to implement is `performCopy(forCollectionView:with:)`. Add the following code to the `UICollectionViewDropDelegate` extension:

```
func performCopy(forCollectionView collectionView: UICollectionView,  
with coordinator: UICollectionViewDropCoordinator) {  
    let destinationIndexPath = coordinator.destinationIndexPath ??  
    IndexPath(item: 0, section: 0)  
  
    for item in coordinator.items {  
        let dragItem = item.dragItem  
        guard dragItem.itemProvider.canLoadObject(ofClass: UIImage.self)  
        else { continue }  
  
        let placeholder =  
        UICollectionViewDropPlaceholder(insertionIndexPath:  
destinationIndexPath, reuseIdentifier: "GalleryCollectionItem")  
        let placeholderContext = coordinator.drop(dragItem, to:  
placeholder)  
  
        dragItem.itemProvider.loadObject(ofClass: UIImage.self) { [weak  
self] item, error in  
            DispatchQueue.main.async {  
                guard let image = item as? UIImage else {  
                    placeholderContext.deletePlaceholder()  
                    return  
                }  
  
                placeholderContext.commitInsertion { indexPath in  
                    self?.images.insert(image, at: indexPath.row)  
                }  
            }  
        }  
    }  
}
```

Take a close look at `UICollectionViewDropPlaceholder` in this snippet. This class was introduced in iOS 11, and it is used to add temporary items to `CollectionView`. Because it might take a little while to load data from an item provider, you need a mechanism to update the UI while you're loading data. This is the goal of using a placeholder. When you call `drop(_:to:)` on `coordinator`, you receive a placeholder context. You use this context to either remove the placeholder if loading data from the item provider failed, or to commit the insertion if it succeeds. Once it has succeeded and you commit the insertion, you must make sure to update the collection's data source by adding the image to the image array. Otherwise, your app could crash due to data-source inconsistencies.

Since a placeholder is not part of your `CollectionView` data source, it is essential that you proceed with caution if you have a placeholder present in your `CollectionView`. For instance, your placeholder will be gone when you reload `CollectionView` before committing or removing the placeholder.

Lastly, add the following line to the `viewDidLoad` of `ViewController` to set the collection view's drop delegate:

```
collectionView.dropDelegate = self
```

At this point, you should be able to create a very nice implementation of drag and drop in your apps. However, there is more to learn on the topic since you can customize many aspects of how drag and drop works for your app.

Customizing the drag and drop experience

Sometimes you will find yourself working on an app where the default implementations don't work for you. For instance, any application's drop delegate can propose a move action instead of a copy action. However, it's possible that you don't want to support this. You can restrict the allowed proposals by implementing `dragInteraction(_:sessionAllowsMoveOperation:)`. If you only want to allow copy operations, you can return `false` from this method. Another restriction you can enable through a delegate method is `dragInteraction(_:sessionIsRestrictedToDraggingApplication:)`. If you return `true` from this method, users won't be able to drag content from your app to another app.

Other methods on both the drag and the drop delegates are related to monitoring the state of the drag or drop session. Imagine that your app supports move proposals. When the user decides to move one or more objects from your app to another, you'll need to update the user interface accordingly once the drop is finished. You can implement `dragInteraction(_:sessionDidTransferItems:)` to implement this or perform any other cleanup that you might want to do after a successful drop session. For instance, you could show an alert to let the user know that data was transferred successfully.

In addition to logic, you can also provide custom previews and perform animations alongside the animations performed by the drag and drop sessions. All methods for this are defined in the `UIDropInteractionDelegate` and `UIDragInteractionDelegate` protocols as optional methods. In many cases, you won't need to implement many customizations for drag and drop. If you do find yourself wanting to customize part of the drag and drop experience, the chances are that a quick glance at the available methods on the delegate protocols will guide you in the right direction.

Summary

This chapter showed you how to implement a smooth drag and drop experience for your users. Drag and drop has limited functionality on the iPhone, but this doesn't stop it from being a powerful feature that can be used to support the dragging of contents within an application or to reorder a `UICollectionView`. Collections and tables have received special treatment because they have their own delegate methods and enable you to easily access the cell that is selected by the user for dragging.

While drag and drop might seem complicated at first glance, Apple did a great job of containing this complexity in a couple of relatively simple delegate methods that you can implement. A simple drag and drop implementation only requires you to implement fewer than a handful of methods, which is quite impressive for such a powerful feature!

In the next chapter, you will perform some extra work on the **MustC** app that you have worked on in previous chapters. You will enhance it by integrating it with iOS' powerful Spotlight search database.

Questions

1. What devices support the full drag and drop experience?
 - a) Only iPhone.
 - b) Only iPad.
 - c) Both iPhone and iPad.

2. What kind of object do you have to provide to start a drag session?
 - a) A `UIDragItem` object.
 - b) A `UIDragInteraction` object.
 - c) An `NSItemProvider` object.
3. How can you check whether your app can handle a certain drop action?
 - a) By checking the type of object that is associated with the drop action.
 - b) By limiting the type of objects that can be used in your drag session.
 - c) You receive a special error type if the app failed to handle the drop.
4. How is the drag and drop experience limited on iPhone?
 - a) You can only start drag sessions on iPhone.
 - b) Your app can only implement drop interactions on iPhone.
 - c) Drag and drop is limited to the currently-open app.
5. What do you have to do to support drag and drop on an iPhone?
 - a) You must manually enable the drag and drop interactions.
 - b) You need a special capability.
 - c) You need to add an `Info.plist` entry.
6. How are the drag and drop delegate protocols for collection views different from the regular drag and drop delegate protocols?
 - a) They are the same, only with different names.
 - b) The collection view versions of the delegate methods receive an index path.
 - c) They are called at different times than the regular delegate methods.

Further reading

- The Drag and Drop developer documentation: https://developer.apple.com/documentation/uikit/drag_and_drop

25

Improved Discoverability with Spotlight and Universal Links

Many users of macOS and iOS love one feature in particular: Spotlight search. Spotlight keeps an index of all files, contacts, contents, and more on your Mac, and it's been in iOS too for a while now. With iOS 9, Apple made it possible for developers to index the contents of their apps, enabling users to discover app contents right from the Spotlight search they know and love. Ever since opening up Spotlight on iOS to developers, Apple has been pushing to make Spotlight better, more relevant, more impactful, and more helpful to users. In this chapter, we'll explore what Spotlight can do for your apps and how you can make use of the `CoreSpotlight` APIs to provide your users with a fantastic search experience that helps them find and navigate your app.

This chapter covers the following topics:

- Understanding Spotlight search
- Adding your app contents to the Spotlight index
- Handling search-result selection
- Adhering to Spotlight best practices
- Increasing your app's visibility with Universal Links

Let's not waste any time and jump right into Spotlight!

Understanding Spotlight search

If you have a Mac, which you most likely do, you have probably used the blazingly fast Spotlight search feature on it. Furthermore, if you love this feature on your Mac, you have probably used it on iOS as well. Spotlight is a highly optimized search engine that enables you to find content throughout your device.



If you haven't used Spotlight, try swiping down on your home screen on iOS and use the search box. Or, on your Mac, press *command + space* to open the Spotlight search dialog.

Spotlight has been around since iOS 3. However, it wasn't until iOS 9 that Apple opened up Spotlight to developers. This means that starting with iOS 9, Spotlight indexes everything from web results, apps, app store results, and any content you add to the Spotlight index yourself. Spotlight presents search results with an image, a title, and some extra information, and sometimes extra buttons are shown to make a phone call or to start turn-by-turn navigation to an address.

The results that are displayed by Spotlight are a mix of public or online content and private results. If your app has indexed items that are publicly available, Apple will make them visible to all users who have your app installed as long as enough people interact with this particular searchable item.

A great example of public content for almost any app is the main navigation items in an app. Imagine you have an app that has a couple of tabs in `UITabBarController`. Any user that downloads your app and opens it can immediately access each tab if they want to. This means that these tabs are public; any user of your app can access these tabs.

Your app can index these tabs using `CoreSpotlight` and mark them as publicly available. When you mark an item as public, it will not be globally visible right away. Behind the scenes, `CoreSpotlight` indexes the item as it usually would when indexing a private record. Whenever a user selects this particular item from a list of Spotlight search results, it sends an irreversible hash of the item to Apple's servers. If Apple receives the same hash over and over again, a certain threshold will be reached. Once this threshold is reached, the indexed item is pushed to all people who have your app installed. If your indexed item has a public web resource associated with it, your publicly searchable item will even be shown to people who don't have your app installed because the content is available on the web. If a user selects this item but doesn't have your app installed, they will be taken to your website instead to view the requested content there.

The fact that public results in Spotlight are kept private for a while helps to make sure that everything your users see when they use Spotlight is relevant. Imagine that any app developer could mark any content as public and every time a developer does this, the item is pushed directly to all users. This would pollute Spotlight, and the entire feature would become virtually worthless. Spotlight's goal is to provide relevant and interesting results. The threshold for interactions makes sure that enough people consider the displayed item to be appropriate for their search query, which means that lousy indexing or spam doesn't have a chance to thrive.

Another reason this threshold exists is to protect your user's data. It's perfectly fine to add personal and private data to Spotlight. The standard mail app does this so users can search their email messages using Spotlight. These email messages should not be publicly indexed so they shouldn't be marked as such. Mistakes happen though, and in case you do end up marking private data as public by accident, there is no way this specific item will show up for other users because the public result threshold for the item will never be reached since only one user has interacted with the Spotlight search result.

Excellent Spotlight visibility can boost usage of your apps, especially since users don't even need to have your app installed to see results for your app as long as there is a web resource available for your indexed contents. Apple has worked hard to ensure that Spotlight is fast and provides a seamless experience for your users. This means that users won't notice whether a result comes straight from their device or whether they see a result because it was publicly indexed.

You might have noticed that Safari sometimes shows results for in-app content. These results come from Spotlight. Whenever you add a web resource to an item you index, Safari will show this resource if it matches a user's query in the search bar. In this chapter, you will learn all about optimally indexing your app's contents; so, if you have an app that has a website that mirrors its contents, you'll know how to maximize your presence in search results for both Spotlight and Safari.

Adding your app contents to the Spotlight index

If you have ever worked on a website, you must have heard something about **Search Engine Optimization (SEO)**. More importantly, you will know that any website you create and publish is indexed by several search engines. All you have to do is make sure that you write semantic and structured HTML markup and any web spider will understand what your website is about and what parts of it are more important.

Search engines, such as Google, have indexed billions of web pages based on their contents and semantic markup.

Apps tend to be a little less neatly structured, and crawling them is a lot harder, if not impossible. There is no formal way to figure out what content is on screen and what this content means. Also, more importantly, a lot of content you'd want to index is only available to users who have logged in or created content of their own.

This is why Apple decided that the developers themselves probably know their app's contents best and should be in charge about how, when, and why a particular content is indexed. Even though this does put a little bit of manual burden on the developers, it gives them a considerable advantage over the automatic indexing that's done on the web. Since developers are in control, they can decide precisely which content matters most to specific users. As you'll soon see, you can index content based on the screens a user visits, which means that you will index those pages that your user may want to visit again.

Even more important than being in control is the ability to index private contents safely. The web is limited to indexing public content. If you use online email software to check your inbox or if you have an online project-management tool, you must rely on the internal search functions inside the web page for these tools. You won't find your emails or projects through a regular search query. With Spotlight indexing, your users can do just that: search through the content they own on their device. Indexing private content is secure because the data is not made available to other apps and you can't accidentally make one user's private data visible to other users due to the public indexing threshold mentioned earlier.

So how exactly do developers take control? That's the question that's answered next. You will learn about the following methods that Apple came up with to index app contents:

- NSUserActivity
- CSSearchableItem
- Universal Links

Indexing your app through user activity

As part of a feature set called **Continuity**, Apple launched Handoff in iOS 8. **Handoff** allows users to start an activity on one device and then continue it on another. In iOS 9, Apple introduced the ability to index these user activities in Spotlight.

There are several advantages to this because apps that support Handoff hardly need to do anything to support Spotlight indexing and vice versa. So, even though this chapter is all about search and Spotlight, you will also learn something about enabling Handoff for your app.

The philosophy behind user activities is that whenever your user interacts with your app, you create an instance of `NSUserActivity`. For Spotlight, these activities revolve solely around viewing content. Any time your user looks at a piece of content in your app is an excellent time to create a user activity and have Spotlight index it. After the user activity is added to the index, the user will be able to find it through Spotlight; when they tap on it, your app can take the user straight to the relevant section of the app, allowing the user to resume their activity.

In the previous chapters, you worked on an app called **MustC**. The **MustC** app collects data about people and their favorite movies. The app also keeps track of a rating for each of the movies. This content is great to index in Spotlight, so let's add some indexing to it.

A few additions were made since the last time you worked on this app. The app now contains a tab bar. There is a tab for the list of family members, and there is one that lists all of the movies that are added to the app. Selecting a movie will display a list of family members that have added this movie to their favorites.

An app such as **MustC** is a great candidate for indexing. You can add the separate family members, the tabs from the navigation bar, and the movies to Spotlight to make them searchable from anywhere within iOS. You will start off with the most straightforward content to track. You will add the family members and movie tabs to Spotlight when a user visits them.

To do this, you will use user activities, so you should create an activity whenever a user opens one of the two tabs. There are two places where you could implement the user activities:

- `viewDidAppear()`
- `viewDidLoad()`

If you create the user activity in `viewDidAppear`, the user activity is pushed every time the user switches tabs or navigates back to the root view controller from another view controller. Even though it doesn't cost much to index a user activity, it seems like it's overkill to index a tab every single time a user sees it.

Even though this train of thought isn't wrong, the best way to go about indexing tabs in Spotlight is actually to create the activity in the `viewDidAppear()` method. If you put this logic in `viewDidLoad()`, it will only be executed once even though the idea of user activities is that they describe each action a user performs. In this case, inserting the same activity over and over again is the desired behavior because it accurately reflects what the user is doing inside the app.

When your app implements user activities, you should always specify them by adding an `NSUserActivityTypes` array to your app's `Info.plist` that contains the user activities your app will handle. Add this key to `Info.plist` for **MustC** and add a single item name, `com.familymovies.openTab`, to the activity types array.

Let's take a look at some code that indexes the family members tab. You should add it to `FamilyMembersViewController`:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    let userActivity = NSUserActivity(activityType:
"com.familymovies.openTab")
    userActivity.title = "Family Members"
    userActivity.isEligibleForSearch = true
    userActivity.isEligibleForPublicIndexing = true
    self.userActivity = userActivity
    self.userActivity?.becomeCurrent()
}
```

The preceding code shows how to create a straightforward user activity. The activity you just created only has a title because there isn't much else to associate with it. The most important thing to note is the `isEligibleForSearch` property. This property tells the system that the user activity that is about to be set as the current activity should be indexed for searching. Other similar properties are `isEligibleForHandoff` and `isEligibleForPublicIndexing`. You used the `isEligibleForPrediction` property in Chapter 20, *Streamlining Experiences with Siri*.

It's a great idea for the activity you just created to be eligible for public indexing, so that property is set to `true`. Doing this will make the activity show up in the search results of a lot more people, given that enough people interact with it. Marking an activity as eligible for handoff allows a user to continue an activity they started on one device on another device. Since this app only works on iOS and you don't need to take multiple users with multiple devices into account, you don't have to set this property to `true`.

Finally, the user activity you just created is set as the current activity. This makes sure that iOS registers the activity and adds it to the Spotlight index. It won't be made available publicly right away due to the threshold of people that interact with this activity that must be reached before Apple will push it to all users. Even though it won't be indexed publicly, it will appear in search results locally for the current user.

If you build and run your application, the family members tab should be the first tab to appear. This means that a user activity for that view is created and indexed immediately. After opening the app, go ahead and open Spotlight by swiping down on the home screen and perform a search for the word **family**.

You'll notice that the activity you just added is listed under the **MUSTC** header in the Spotlight search results, as shown in the following screenshot:



Pretty neat, right? You were able to add a simple entry in Spotlight's search index with very minimal effort. You should be able to add a similar implementation to the **MustC** app to have it index the `Movies` page. Go ahead and add a modified version of the snippet you added to `FamilyMembersViewController` to `MoviesListViewController`.

Now that both tabs show up in Spotlight, how do you make sure that the correct tab is opened when the user selects a result from Spotlight? The answer lies in one of the `AppDelegate` methods. When your app is brought to the foreground because a user selected your app as a Spotlight search result, the `application(_:continueUserActivity:restorationHandler:)` method is called. This is the same method that gets called when a user executes one of their Siri Shortcuts, so if you need a little refresher on how this method works, make sure to go back to Chapter 20, *Streamlining Experiences with Siri*.

Currently, there are only two Spotlight entries that the user can select. They either want to see the family members tab or the movies tab. The implementation of `application(_:continueUserActivity:restorationHandler:)` should inspect the user activity that it received to determine which tab should be displayed. Once this is established, a reference to the app's `UITabBarController` should be obtained and the correct tab should become active.

Since each tab contains a navigation controller, you should always make sure to pop the relevant navigation controller to its root view controller. This is needed because the user might have been looking at a detail view controller, and the app would show this detail view controller instead of the root view controller if you don't ensure that the navigation controller pops to its root view controller. Add the following code to `AppDelegate` to add this functionality:

```
func application(_ application: UIApplication, continue userActivity: NSUserActivity, restorationHandler: @escaping ([UIUserActivityRestoring]?) -> Void) -> Bool {
    // 1
    guard let tabBar = window?.rootViewController as? UITabBarController
        else { return false }

    // 2
    let tabIndex: Int?
    if userActivity.title == "Family Members" {
        tabIndex = 0
    } else if userActivity.title == "Movies" {
        tabIndex = 1
    } else {
        tabIndex = nil
    }
}
```

```
}

guard let index = tabBarIndex
else { return false }

// 3
guard let navVC = tabBar.viewControllers?[index] as?
UINavigationController
else { return false }

// 4
navVC.popToRootViewControllerAnimated(animated: false)
tabBar.selectedIndex = index

return true
}
```

The preceding code first obtains a reference to the tab bar controller. After doing this, the second step is to determine the correct tab index for the selected user activity. Then, the navigation view controller associated with the tab bar is grabbed. The fourth and final step is to pop the navigation controller to its root view controller and to make the correct tab bar index the currently-selected tab bar item.

At this point, two of the screens in **MustC** are indexed in Spotlight. While this is great, there are a couple more screens that can be implemented. Doing this will make the implementation for

`application(_:continueUserActivity:restorationHandler:)` more complicated than it is now.

Manually creating and resuming user activities for each screen in an app is tedious and involves quite a lot of boilerplate code. You can solve this problem by utilizing an activity factory. A typical pattern in apps is to use a specific helper object called a **Factory**. The sole purpose of a factory is to act as an object that is responsible for creating objects of a particular type. This dramatically reduces boilerplate code and increases maintainability. Create a new file called `IndexingFactory.swift` in the `Helpers` folder and add the following implementation:

```
import Foundation

struct IndexingFactory {
    enum ActivityType: String {
        case openTab = "com.familymovies.openTab"
        case familyMemberDetailView =
"com.familymovies.familyMemberDetailView"
        case movieDetailView = "com.familymovies.movieDetailView"
    }
}
```

```
    static func activity(withType type: ActivityType, name: String,  
makePublic: Bool) -> NSUserActivity {  
    let userActivity = NSUserActivity(activityType: type.rawValue)  
    userActivity.title = name  
    userActivity.isEligibleForSearch = true  
    userActivity.isEligibleForPublicIndexing = makePublic  
  
    return userActivity  
}  
}
```

Note that this object contains an enum with three different activity types. Don't forget to add the two new activity types to the `NSUserActivityTypes` list in the app's `Info.plist`. `IndexingFactory` has a single static method in it. This method takes a couple of configuration arguments and uses these to create and return a new user activity instance. Let's see this factory in action. You can add the following code to the `MoviesViewController` class:

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
  
    guard let familyMemberName = familyMember?.name  
    else { return }  
  
    self.userActivity = IndexingFactory.activity(withType:  
.familyMemberDetailView, name: familyMemberName, makePublic: false)  
    self.userActivity?.becomeCurrent()  
}
```

The preceding snippet is a lot smaller than creating a user activity from scratch in every `viewDidAppear`. Also, if you decide to make changes to the way user activities are created and configured, it will be easier to refactor your code because you will only have to change a single method. Add a comparable implementation of the preceding method to the `MovieDetailViewController` class so movie pages will also be indexed by Spotlight when the user visits them.

This wraps up simple indexing with `NSUserActivity`. Next up, you will learn how to index content with the `CSSearchableItem` class and how you can use it to index content the user hasn't seen yet. You'll also see how you can associate more sophisticated data with your searchable items and how Spotlight handles updating and re-indexing contents.

Since `AppDelegate` can't handle opening user activity types that should take the user straight to a family member or movie, it's a good exercise for you to try implementing this logic yourself. For a correct implementation, you'll need to add a new `find` method to the `Movie` and `FamilyMember` classes, and you'll have to instantiate view controllers straight from a storyboard. If you get stuck implementing this, don't hesitate to look at the finished code for this chapter in the code bundle.

Indexing with `CSSearchableItem`

Currently, Spotlight indexing works and users can find any content they have seen before. It's even possible to select results from the Spotlight index and have the app open on the correct page. If you've taken on the challenge of implementing the handling of detail pages, your app should be able to handle the continuation of any activity that was indexed. Wouldn't it be cool if we could be a bit more proactive about indexing though? Ideally, the app would index any new `Family Members` or movies as soon as the user adds them.

This is what `CSSearchableItem` is good at. The `CSSearchableItem` class enables you to index content that the user might not have seen before. Indexing `CSSearchableItem` instances is pretty straightforward. The steps involved are similar to how you could index user activities. To index a searchable item, you should create an instance of `CSSearchableItem` and provide it with attributes that describe the indexed item. These attributes are encapsulated in an instance of `CSSearchableItemAttributeSet`.

Containing information in `CSSearchableItemAttributeSet`

When you create an attributes set, it's important that you populate it correctly since it describes almost all of the important information about your item for Spotlight. You can set a title, content description, a thumbnail image, keywords, even ratings or phone numbers, GPS information, and much, much more. For a full overview of what's possible, refer to the `CSSearchableItemAttributeSet` documentation. Every time you are about to create a new item that can be indexed, you should take a look at the documentation to make sure you don't miss any attributes.

The better use you make of the available attributes, the more effectively your content can be indexed and the higher your app will rank. Therefore, it's worth putting slightly more time and effort into your search attributes because getting it wrong can be a costly mistake, especially considering the available documentation. At a minimum, you should always try to set `title`, `contentDescription`, `thumbnailData`, `rating`, and `keywords`. This isn't always relevant or even possible for the items you're indexing, but whenever possible make sure that you set these attributes.

You may have noticed that the `NSUserActivity` instances you indexed in the app didn't receive any special attributes. You just set a name and some other basic information, but you didn't have to add a description or a rating to any of the indexed objects. If you index user activities in your applications, it's worth noting that user activities can and should have attributes associated with them. All you need to do is set the `contentAttributeSet` property on the user activity. After you implement indexing through `CSSearchableItem`, you should shortly revisit user-activity indexing to make the indexed item richer and also to make sure that `CoreSpotlight` understands that the user activities and searchable items point to the same underlying index in Spotlight.

When you index items through multiple methods, it's inevitable that you run into data duplication. The **MustC** app indexes every visited screen. So, if a user visits the details page of a movie, a user activity for that movie should be created. However, you also want to index movies as they are created by the user. To avoid duplicate results in Spotlight search, `relatedUniqueIdentifier` should be added to the attributes set. Setting this attribute on a user activity makes sure that Spotlight doesn't add duplicate entries for items with the same identifier.

Let's expand `IndexingFactory` with two methods that can generate attribute sets for searchable items. Putting this functionality in `IndexingFactory` as a separate method is a good idea because, if it is set up correctly, these methods can be used to generate attributes for both user activities and searchable items. This avoids code duplication and makes it a lot easier to add or remove properties in the future. Add the following methods to the `IndexingFactory` struct. Don't forget to import `CoreSpotlight` at the top of the file:

```
static func searchableAttributes(forMovie movie: Movie) ->
    CSSearchableItemAttributeSet {
    do {
        try movie.managedObjectContext?.obtainPermanentIDs(for: [movie])
    } catch {
        print("could not obtain permanent movie id")
    }
}
```

```
let attributes = CSSearchableItemAttributeSet(itemContentType:  
ActivityType.movieDetailView.rawValue)  
attributes.title = movie.title  
attributes.contentDescription = "A movie that is favored by  
(movie.familyMembers?.count ?? 0) family members"  
attributes.rating = NSNumber(value: movie.popularity)  
attributes.identifier =  
"(movie.objectID.uriRepresentation().absoluteString)"  
attributes.relatedUniqueIdentifier =  
"(movie.objectID.uriRepresentation().absoluteString)"  
  
return attributes  
}  
  
static func searchableAttributes(forFamilyMember familyMember:  
FamilyMember) -> CSSearchableItemAttributeSet {  
do {  
try familyMember.managedObjectContext?.obtainPermanentIDs(for:  
[familyMember])  
} catch {  
print("could not obtain permanent family member id")  
}  
  
let attributes = CSSearchableItemAttributeSet(itemContentType:  
ActivityType.familyMemberDetailView.rawValue)  
attributes.title = familyMember.name  
attributes.identifier =  
"(familyMember.objectID.uriRepresentation().absoluteString)"  
attributes.contentDescription = "Family Member with  
(familyMember.favoriteMovies?.count ?? 0) listed movies"  
attributes.relatedUniqueIdentifier =  
"(familyMember.objectID.uriRepresentation().absoluteString)"  
  
return attributes  
}
```

For both the `Movie` and `FamilyMember` objects, a set of attributes is created. This set meets Apple's recommendations as closely as possible. The objects don't have any thumbnail images or keywords that can be added to the attributes set other than the movie title or the name of a family member. Adding these to the keywords is kind of pointless because the title in itself is essentially a keyword that the item will match on.

To create an attribute set for a movie or family member, you only have to call the method that matches the object you want to index and you're good to go. Beautiful, convenient, and simple.

Adding CSSearchableItem instances to the search index

In the **MustC** application, the goal is to add family members and movies to the search index as soon as the user adds them. A factory method that creates the `CSSearchableItemAttributeSet` instance that describes the item that should be indexed is already provided. However, you can't directly add these to the index. To add information to the search index manually, you need instances of `CSSearchableItem`. Every searchable item requires a unique identifier and a domain identifier.

The unique identifier is used to identify an indexed item. It's essential that you set this value to something that is unique because otherwise, Spotlight will overwrite the entry with something else that has the same identifier or you'll get duplicate entries if you combine user activities and search items like we're doing for **MustC**.

The domain identifier functions as a namespace. Within any given namespace, all entries must be unique and are identified through their unique identifier. These identifiers only have to be unique within their namespace. Think of this as streets and addresses. In a particular area, every street name is unique (domain, namespace). On each street the house number is unique (unique identifier), but the same number can occur on different streets. The domain identifier for your Spotlight entry is not only used to identify entries uniquely, it's also used to perform specific batch actions on the index, such as deleting all indexed items from a particular domain.

The domain identifier, unique identifier, and the attributes together make up a searchable item. The following code adds factory methods to `IndexingFactory` that will make it simple for the app to add items to the search index:

```
enum DomainIdentifier: String {
    case familyMember = "FamilyMember"
    case movie = "Movie"
}

static func searchableItem(forMovie movie: Movie) -> CSSearchableItem
{
    let attributes = searchableAttributes(forMovie: movie)

    return searchableItem(withIdentifier:
        "(movie.objectID.uriRepresentation().absoluteString)", domain: .movie,
        attributes: attributes)
}

static func searchableItem(forFamilyMember familyMember: FamilyMember)
```

```
-> CSSearchableItem {
    let attributes = searchableAttributes(forFamilyMember: familyMember)

    return searchableItem(withIdentifier: "(familyMember.objectID)",
domain: .familyMember, attributes: attributes)
}

private static func searchableItem(withIdentifier identifier: String,
domain: DomainIdentifier, attributes: CSSearchableItemAttributeSet) ->
CSSearchableItem {
    let item = CSSearchableItem(uniqueIdentifier: identifier,
domainIdentifier: domain.rawValue, attributeSet: attributes)

    return item
}
```

The preceding code defines an enum that contains domain identifiers for items that should be indexed. Note that

`searchableItem(withIdentifier:domain:attributes:)` is marked as private. This is done to make sure anybody using the factory helper has to use `searchableItem(forFamilyMember:)` and `searchableItem(forMovie:)` instead. These methods are simpler to use because they only take a family member or a movie and if you use only these methods, you can rest assured that you only insert consistently configured items into the Spotlight index.

Now that everything is set up for indexing new data, let's begin with indexing family members as soon as the user creates them. Add the following code as the implementation for the `.insert` case in

`controller(_:didChange:at:for:newIndexPath:)` in `FamilyMembersViewController`. Make sure that you import `CoreSpotlight` at the top of your file:

```
guard let insertIndex = newIndexPath,
      let familyMember = fetchedResultsController?.object(at: insertIndex)
else { return }

let item = IndexingFactory.searchableItem(forFamilyMember:
familyMember)
CSSearchableIndex.default().indexSearchableItems([item],
completionHandler: nil)

tableView.insertRows(at: [insertIndex], with: .automatic)
```

Due to the factory methods you set up before, new items can be added to the search index with just a few lines of code. To insert an item into the search index, an instance of CSSearchableIndex is created, and Spotlight is told to index this item. If needed, a completion-handler can be passed to the index method. This handler is called with an optional error. If the indexing has failed, the error should tell you why Spotlight couldn't index the item, and you can retry indexing the item or take a different action. In the **MustC** app, it is assumed that the indexing succeeds and no error-handling is implemented.

Update the `mangedObjectContextDidChange(notification:)` method in `MoviesViewController` as follows to index new movies:

```
@objc func managedObjectContextDidChange(notification: NSNotification) {
    guard let userInfo = notification.userInfo
        else { return }

    if let updatedObjects = userInfo[NSUpdatedObjectsKey] as?
        Set<FamilyMember>,
        let familyMember = self.familyMember,
        updatedObjects.contains(familyMember) {

        let item = IndexingFactory.searchableItem(forFamilyMember:
            familyMember)
        CSSearchableIndex.default().indexSearchableItems([item],
            completionHandler: nil)

        tableView.reloadData()
    }

    if let updatedObjects = userInfo[NSUpdatedObjectsKey] as? Set<Movie>
    {
        for object in updatedObjects {
            if let familyMember = self.familyMember,
                let familyMembers = object.familyMembers,
                familyMembers.contains(familyMember) {

                let item = IndexingFactory.searchableItem(forMovie: object)
                CSSearchableIndex.default().indexSearchableItems([item],
                    completionHandler: nil)

                tableView.reloadData()
                break
            }
        }
    }
}
```

Right after the movie is added to the current family member, it is added to the Spotlight index. You might argue that the item is added too early because the movie's rating has not been fetched at that point. This is OK because when the context is saved the second time with the ratings attached, the item is added to the index again automatically due to the save notification that Core Data sends.

If you run the app now, you should be able to go into Spotlight right after you add a family member or movie, and you should immediately be able to find the freshly added content in Spotlight. If you search for a movie you just added, you'll notice that you can see how many family members have added a certain movie to their favorites list and the rating a movie has. More importantly, there should be only a single entry for each movie because a unique identifier is used to avoid duplicate entries.

Now that you have implemented `CSSearchableItem` and `NSUserActivity` indexing, you need to make sure that both of these objects are correctly configured so you can safely combine both indexing methods.

Safely combining indexing methods

Since no unique information is associated with the user activities yet, Spotlight can't figure out that a family member that's indexed through a user activity is the same family member that was already inserted as a searchable item. To ensure that Spotlight understands this, you should add two more factory methods that create an activity item for either a family member or a movie with the correct information associated with them. Add the following methods to `IndexingFactory`:

```
static func activity(forMovie movie: Movie) -> NSUserActivity {
    let activityItem = activity(withType: .movieDetailView, name:
        movie.title!, makePublic: false)
    let attributes = searchableAttributes(forMovie: movie)
    attributes.domainIdentifier = DomainIdentifier.movie.rawValue
    activityItem.contentAttributeSet = attributes

    return activityItem
}

static func activity(forFamilyMember familyMember: FamilyMember) ->
    NSUserActivity {
    let activityItem = activity(withType: .movieDetailView, name:
        familyMember.name!, makePublic: false)
    let attributes = searchableAttributes(forFamilyMember: familyMember)
    attributes.domainIdentifier = DomainIdentifier.familyMember.rawValue
    activityItem.contentAttributeSet = attributes
}
```

```
        return activityItem
    }
```

The most important lines to take note of are the ones where a `domainIdentifier` is set on the `attributes` constant. Since iOS 10, developers have been able to associate a `domainIdentifier` with user activities through `contentAttributeSet`. By adding a `domainIdentifier` to the indexed item, searchable items and user activities are unified even more. Update the `viewDidAppear` implementation for `MovieDetailViewController` as follows:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    guard let movie = self.movie
    else { return }

    self.userActivity = IndexingFactory.activity(forMovie: movie)
    self.userActivity?.becomeCurrent()
}
```

We also need to update the `viewDidAppear` method in `MoviesViewController`. You should be able to do this on your own; the code will look similar to the preceding snippet, except you're indexing a family member instead of a movie.

Now that all of your app contents are indexed in Spotlight, it's time to discuss some of the methods Spotlight uses to rate your content and the best practices you should keep in mind when you add your app's contents to Spotlight.

Handling searchable item selection

If a user taps on a search result for one of the items you indexed manually, the `application(_:continue:restorationHandler:)` method is called on `AppDelegate`. This is the same method that's used for user activities, but the internal handling is not quite the same.

In the original code to continue user activities, the user activity's `title` property was used to determine the appropriate action. Since searchable items don't have this property, a different approach will have to be implemented. To decide whether the user activity that is passed to `application(_:continue:restorationHandler:)` is a searchable item or a user activity, you can read the `activity type`. If the user activity was originally a searchable item, the `activity type` should be equal to `CSSearchableItemActivityIdentifier`. If you receive a user activity with this `activity type`, you should take action accordingly.

Update your code in `AppDelegate` as follows and make sure to import `CoreSpotlight`:

```
func application(_ application: UIApplication, continue userActivity: NSUserActivity, restorationHandler: @escaping ([UIUserActivityRestoring]?) -> Void) -> Bool {

    if let identifier =
        userActivity.userInfo?[CSSearchableItemActivityIdentifier] as? String,
        userActivity.activityType == CSSearchableItemActionType {

        return handleCoreSpotlightActivity(withIdentifier: identifier)
    }

    // existing implementation
}

func handleCoreSpotlightActivity(withIdentifier identifier: String) -> Bool {
    guard let url = URL(string: identifier),
          let objectID =
        persistentContainer.persistentStoreCoordinator.managedObjectID(forURIRepresentation: url),
          let object = try?
        persistentContainer.viewContext.existingObject(with: objectID)
        else { return false }

    if let movie = object as? Movie {
        return handleOpenMovieDetail(withName: movie.name!)
    }

    if let familyMember = object as? FamilyMember {
        return handleOpenFamilyMemberDetail(withName: familyMember.name!)
    }
    return false
}
```

The updated version of `application(_:continue:restorationHandler:)` checks whether it received a searchable item. If it did, a special method is called. This method uses the persistent store to convert the string identifier to a managed object ID which is then used to ask the managed object context for the corresponding object. If all of this succeeds, we attempt to cast the fetched object to either a movie or a family member, and if this succeeds, one of the existing handlers is called. If none of the casts succeed, it was not possible to continue the activity, so the method returns false.

Note that this implementation assumes that you did the exercise that was proposed earlier in the chapter so `handleOpenFamilyMemberDetail(withName:)` and `handleOpenMovieDetail(withName:)` do not exist in the project. Refer to this chapter's code bundle to see a suggested implementation for both of these methods.

Understanding Spotlight best practices and ratings

If you implement Spotlight indexing in your app, it's beneficial for both you and your users if you adhere to best practices. Sticking with the best practices will positively affect your ratings and can ultimately drive more users to your app.

There are two best practices that you should always follow if possible. Doing so will give your app great advantages over other apps that might not stick to best practices. The two best practices you should always try to follow are:

- Adding metadata to your web links
- Registering as an indexing delegate

Adding metadata to your web links

In the next section about Universal Links, you will learn what Universal Links are and how you should implement them in your application. Enabling Universal Links benefits your Spotlight search ranking and visibility because Spotlight can associate web resources that Apple's bots can crawl with the items you add to the Spotlight index. To associate Universal Links and your locally indexed items with each other, you should set the `contentUrl` that matches the web resource for a local item on your search attributes objects.

Doing so allows Apple to display search results for your app as public, and it can even show your Universal Links in Safari's search results. This means that it will be even easier for your users to discover and use your app. More importantly, if you associate a `contentUrl` with an indexed item, your app is also shown to people who don't have your app installed. That fact alone should persuade you to make sure that your app has online resources that mirror the public content in your app.

Once you've decided to add content URLs to your content, it's essential that you add proper, structured metadata to your web pages. If your web page has well-formatted metadata available, Spotlight will index your app's contents even if you don't manually add it. This is because Apple continuously crawls the marketing and support URLs you advertise within the App Store.

For Apple's bot to be able to discover the URLs your app can handle, it checks whether there is a Smart App Banner implemented on your web page. The Smart App Banner is visible to people who visit your website. If they already have your app installed, the banner will prompt the user to view this content in your app. If you've added a URL to the banner that your app can handle, Apple's bot can discover this link and associates it with your app.

You should always make sure that you add useful metadata to your web content. When you manually index content, you can set properties on the indexed item's attributes set to represent your data. On the web, you can't create an attribute set, so you need to embed the metadata in your page's HTML markup. If your web page contains Open Graph metadata, Apple's bots will understand how to read that metadata. Even though Open Graph metadata is a great start, Apple recommends that you implement richer and more fine-grained data for your websites.

To do this, it's recommended that you implement metadata as specified in the schema.org standards. These standards provide definitions for a wide range of different entities. For instance, you can express a product rating, pricing, or a business address through `schema.org` definitions of these entities.

If you've implemented metadata for your app, or if you want to make sure that Spotlight can index your contents, you can use Apple's verification tools for searching. Go to <https://search.developer.apple.com> to paste in your website's URL and verify that everything can be indexed as you expected.

Having a complete set of metadata positively impacts your rating. An obvious reason is that it's better because more information means more matches for a search query. However, a different reason is that, if a user sees a search result that provides a lot of information at a glance, it's easier for them to decide whether a specific result is relevant to them.

This quick assessment of relevancy up front makes sure that whenever a user selects a result for your app, they know for sure that they want to interact with that item. Apple measures this form of engagement to ensure that results that are tapped often and are used longer are pushed to the top.

A topic that is directly related to this is the concept of keyword stuffing. When optimizing for a search, it can be tempting to stuff a lot of keywords into your contents. Doing this will not positively impact your ratings in Spotlight. The reasoning behind this is that keyword-stuffed results will often look like spam. The preview won't be a coherent preview of contents, but it will look as if somebody tried to put as many relevant words together as they could. A result that looks poor is less likely to be tapped by a user and will be pushed down in the rankings eventually.

If users do end up tapping a keyword-stuffed item, it's unlikely that they will find what they were looking for. Often, this means that the user exits your app after just a couple of seconds, and this negatively impacts your result rating.

In other words, add as much metadata as you reasonably can, but ensure that any data you add is relevant, authentic, and exists in your app. A good preview in Spotlight and a matching experience in your app is the best way to for apps to rank highly in Spotlight.

Registering as an indexing delegate

An item you add to the Spotlight index today might change over time. Some items you add to Spotlight could even have an expiration date if you've set the `endDate` property on the item attributes. Therefore, it's important that you register your app as an indexing delegate.

The indexing delegate is used for a single purpose: re-indexing content. If you've added items that expire over time, Spotlight will ask you to index them again to make sure that the expiration date is still correct. The indexing delegate is also called if something goes wrong with the index itself; for example, if all pieces of data are lost or if the search index becomes corrupted.

The indexing delegate has two required methods:

```
searchableIndex(_:reindexAllSearchableItemsWithAcknowledgementHandler:) and  
searchableIndex(_:reindexSearchableItemsWithIdentifiers:acknowledgmentHandler:).
```

The first method is called if the index has been lost and everything should be indexed. It's up to you to figure out how to index your entire app. In the **MustC** app, this would probably mean fetching all family members and indexing them. You could use the `movies` relationship on the family members to loop over all of the movies that are stored in the database and index them accordingly. Alternatively, you could query both entities separately and index them separately.

The second indexing delegate method receives an array of identifiers that need to be indexed and is used to re-index a specific set of items. There are several reasons for spotlight to call this method, for instance, if an item is about to expire.

After you index the required items, you should call the `acknowledgementHandler` that the `delegate` method has received. This will make sure that Spotlight understands that you have successfully executed the tasks you needed to perform. To set the indexing delegate, you set the `indexDelegate` of the search index. For most apps, the searchable index is accessed through `CSSearchableIndex.default()`.

Now that you know everything about adequately implementing Spotlight indexing, let's see how you can step up your indexing game a little bit more by implementing Universal Links.

Increasing your app's visibility with Universal Links

A Universal Link is very similar to a deep link. **Deep links** allow apps to link users straight into a certain section of an application. Before Universal Links, developers had to use a custom URL scheme to create their deep links.

You might have seen a URL with a custom scheme in the past. These URLs are easily recognized and look as follows:

```
mustc://FamilyMember/jack
```

It's evident that this isn't a regular web URL because web URLs start with a scheme of either `http://` or `https://`. An application can register itself as capable of opening URLs with a certain scheme. So, the **MustC** app we've been working on could manifest itself as a handler of `mustc://` URLs.

However, there are a couple of downsides to this approach. First and foremost, this URL isn't shareable at all. You can't send this URL to any friends that don't have the same app installed. If you were to send this URL to somebody and they didn't have the corresponding app installed, they wouldn't be able to open this URL. This is inconvenient because, for others to access the same content, assuming the content is publicly available on the web, users would have to share a different URL that points to the website. But sharing a link to the website usually means that the content is shown in Safari instead of the app, even if it's installed.

Another problem with custom URL schemes is that any app can register as being capable of opening a certain URL scheme. This means that you could create an application that registers as being capable of opening URLs with any scheme you can come up with and unfortunately, iOS offers the users no control over which application opens what URL scheme.

Universal Links were introduced in iOS 9 to solve all of the problems that exist with custom URL schemes and more. First of all, a Universal Link looks identical to a regular web link. A Universal Link is similar to a regular web link. If you've found a great news article on the web and you share it with somebody who has installed the app that belongs to the news website the link is from, the link will redirect straight to the corresponding app. Safari does not open intermediately; no attempts are made to redirect you from a web page to a custom URL scheme. The user merely is taken from the place where they tap the link, right to the app.

Also, not every app can register as capable of opening a Universal Link. Any app that claims to be able to open a certain Universal Link must be accompanied by the server that hosts the website. This means that, if your app claims to be able to open links from a certain domain, you must own that domain. Apple uses a verification file that you must host on the same domain that your app wants to handle links for, to make sure that your app does not try to open links on behalf of another app or website.

Apart from security benefits, Universal Links also provide a more unified, seamless experience for your users. With Universal Links, Apple didn't just open the door to a better, easier way to link to content inside your app; it also opened up an API that makes it easy for your app and website to share login information securely. Just like tying the links for your app and website together, you can also tie your app to the login credentials stored in Safari for your app. Any user that logs into your website through Safari can automatically be logged into your app.

Now that you're aware of the great features and possibilities of Universal Links, let's see how this all works on the server side.

Preparing your server for Universal Links

Setting up Universal Links on the server side has been made as straightforward as possible by Apple. All you have to do is host a file on your server that can prove the connection between your app and your server. This is done through a file called the `apple-app-site-association` file.

The `apple-app-site-association` file contains a dictionary of information that describes exactly how your app is tied to your site. Let's have a look at an example of an `apple-app-site-association` file. The example illustrates the implementation for a recipe app where users can browse and search recipes:

```
{  
    "applinks": {  
        "apps": [],  
        "details": {  
            "6QA73RGQR2.com.donny.recipes": {  
                "paths": [  
                    "/recipes/*/",  
                    "/search/"  
                ]  
            }  
        }  
    }  
}
```

Let's go over this configuration file bit by bit. Firstly, some mandatory dictionary keys are set up. The `applinks` key tells Apple that this part of the association file applies to Universal Links. Inside this key, an empty `apps` array is defined. This array does not need to have any content in it for a simple integration. The `details` key is a dictionary that contains the actual configuration.

Your app identifier should be added as a key for the `details` dictionary. The prefix you see in front of the app identifier is your team identifier. You can find your team identifier in the Apple Developer portal.

There is also a `paths` array inside the dictionary that's associated to the bundle identifier key. This array specifies all the paths on our website that your app should handle. Imagine that your website's base URL is `https://www.donnysrecipes.com/`. For this URL, `https://` is the scheme, `www.donnysrecipes.com` is the domain, and anything after the trailing `/` is called the path.

The example configuration handles the `/recipes/*/` and `/search/` paths. The `*` in `/recipes/*/` represents a wildcard value that could be anything. In other words, the config is set up to handle URLs such as `https://www.donnysrecipes.com/recipes/10/`, `https://www.donnysrecipes.com/recipes/20/`, or any other URL that looks similar.

The `/search/` path is a bit more interesting. No wildcard values were specified for this path, yet the app will be able to handle URLs such as `https://www.donnysrecipes.com/search/?q=macaroni`. That final part in the URL, `?q=macaroni`, is called the query string and you don't need to specify that you want to match on that because it's not part of the path.

Once you have created and configured your `apple-app-site-association` file, you need to upload it to your server. It's important that you host the association file in a way that makes it accessible on a URL similar to `https://www.donnysrecipes.com/apple-app-site-association`. In other words, the path to this verification file on your server should be `/apple-app-site-association`.

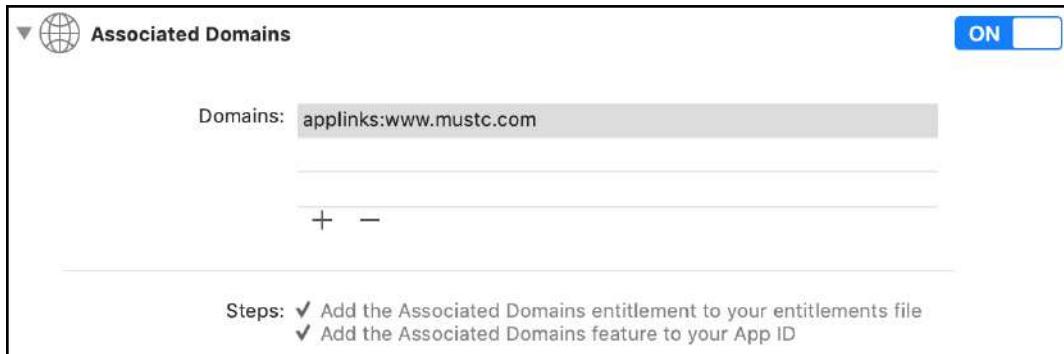
Now that the server is ready to communicate to Apple that your app can open links for your site, it's time to see how you can set up your app to handle Universal Links.

Handling Universal Links in your app

With all parts in place, you can now set your app up to handle Universal Links. In the previous sections of this chapter, a lot of work was done to handle user activities nicely. The setup you created for this will serve as a foundation for handling Universal Links.

Whenever an application is expected to open a Universal Link, the `application(_:open:options:)` method is called on `AppDelegate`. This method receives the URL it's expected to open. If the URL can't be opened, this method is expected to return `false`. If this method does manage to handle the URL, it should return `true`.

Any application that handles Universal Links must have the **Associated Domains** capability enabled in the Capabilities tab. To do this, go to the **Capabilities** tab in your project settings and allow **Associated Domains**. For every domain that you want to handle Universal Links for, you need to create a new `applinks:` entry. The following screenshot shows an example configuration:



When the `application(_:open:options:)` you are expected to map the URL is passed to this method to a view in the application and take the user to that specific view. This process is called routing, and it's a well-known technique throughout programming languages and platforms.

Let's assume that your application received the following URL:

```
https://www.familymoviesapp.com/familymember/dylan/
```

Just by looking at this URL, you probably have a vague idea of what should happen if an application is asked to open this URL. The user should be taken to a family member screen that shows the detail page for Dylan.

When you receive a URL such as this in `AppDelegate`, three separate parts exist on the URL. One is the `scheme`; this property tells you which URL scheme, for example `https://`, was used to navigate to the app. Usually, the scheme isn't relevant to your app unless you're handling multiple custom URL schemes. Second, there is the `host`, for example `example.com`. This property describes the domain that the URL belongs to. Again, this is usually not relevant unless your app handles links from multiple hosts. Finally, there is the `pathComponents` property. `pathComponents` is an array of components that are found in the path for the URL. Printing `pathComponents` for the example URL gives the following output:

```
["/", "familymember", "dylan"]
```

The first component can usually be ignored because it's just /. The second and third components are a lot more interesting. They tell you more about the route that needs to be resolved. Going back to the **MustC** example, you could handle URLs in that app easily with the code that already exists in AppDelegate to navigate to family member and movie pages. The following code shows how:

```
func application(_ app: UIApplication, open URL: URL, options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
    let pathComponents = URL.pathComponents
    guard pathComponents.count == 3
        else { return false }

    switch(pathComponents[1], pathComponents[2]) {
    case ("familymember", let name):
        return handleOpenFamilyMemberDetail(withName: name)
    case ("movie", let name):
        return handleOpenMovieDetail(withName: name)
    default:
        return false
    }
}
```

A switch with powerful pattern-matching is used to check whether the second component in the URL points to a family member or a movie and we parse the third component into a variable called name. If one of these matches, we call the existing methods; if nothing matches, we return false to indicate that we couldn't open the URL.

Earlier, you saw this URL:

<https://www.donnysrecipes.com/search/?q=macaroni>. With pathComponents, you can easily gain access to the path of the URL. But how do you get the final part of the URL? Well, that's a little harder. You can get the query property for a URL, but then you get a single string in the form of q=macaroni. What you want is a dictionary where q is a key and macaroni is a value. The following extension on URL implements a naive method to create such a dictionary:

```
extension URL {
    var queryDict: [String: String]? {
        guard let pairs = query?.components(separatedBy: "&")
            else { return nil }

        var dict = [String: String]()
        for pair in pairs {
            let components = pair.components(separatedBy: "=")
```

```
        dict[components[0]] = components[1]
    }

    return dict
}
}
```

First, the query string is retrieved. Then the string is separated on the & character because multiple key-value pairs in the query string are expected to be separated with that character. The code loops over the resulting array and separates each string on the = character. The first item in the resulting array is used as the dictionary key and the second item is the value. Each key and value are added to a dictionary, and finally, that dictionary is returned. This allows you to get the value for q, shown as follows:

```
URL.queryDict!["q"]
```

Great job! This is all that's needed to implement Universal Links in your app and to enhance discoverability for your users.

Summary

This chapter focused on getting your app indexed with the powerful `CoreSpotlight` framework. You saw how you can use user activities to index items that the user has already seen and how you can use searchable items to index content that the user might not have seen. You also learned how to make use of unique identifiers to ensure that you don't end up with duplicate entries in the search index. We also took a look at continuing user activities from either a user activity or a searchable item.

After you saw how `CoreSpotlight` can index your items, you learned about Universal Links and web content. It's important to think about publishing content on the web because it helps your Spotlight indexing tremendously and it enables Apple to show your Spotlight results in Safari. You also learned about the metadata that you should add to your web pages.

After all this Spotlight content, you were introduced to Universal Links. You learned how they relate to your app's web pages, and how you can use them to get bonus points in the Spotlight index and to improve the user's experience with your app.

In the next chapter, you will learn how you can integrate your apps with iMessage through an iMessage app.

Questions

1. Which objects are used to represent an item that you would like to index in Spotlight?
 - a) `NSUserActivity` and `CSUserActivity`.
 - b) `NSUserActivity` and `CSSearchableItem`.
 - c) `CSSearchableItem` and `CSSearchableItemAttributeSet`.
2. What happens when a user selects a Spotlight search result that points to your app?
 - a) `AppDelegate` is asked to resume a user activity that represents the item.
 - b) `AppDelegate` is asked to handle a URL that represents the item.
 - c) `CSsearchableIndexDelegate` is asked to handle the selected item.
3. How do you add an item to the Spotlight index?
 - a) By passing a `CSSearchableItem` to `CSSearchableIndex` or setting a user activity as the current activity.
 - b) By implementing `CSSearchableIndexDelegate`.
 - c) Only by passing a `CSSearchableItem` to `CSSearchableIndex`.
4. Does it make sense to index the same user activity multiple times?
 - a) No, the item will only appear once in the Spotlight index.
 - b) Yes, the more often an activity is performed in the app, the higher it will rank in the Spotlight search results.
 - c) No, Spotlight will punish apps that add duplicate items.
5. What are Smart App Banners?
 - a) You can add Smart App Banners to web pages to point visitors to your app.
 - b) Metadata objects that you add to your app for better indexing.
 - c) Special items that appear in the Spotlight search results.

6. What are Universal Links?

- a) Shareable web links that open your app if it's installed.
- b) Links with a custom URL scheme that point to an app.
- c) Links with a custom URL scheme that point to a website.

7. What happens when your app is expected to open a Universal Link?

- a) AppDelegate is asked to continue a user activity.
- b) AppDelegate is asked to open the URL.
- c) The link your app should open is passed to the launch arguments dictionary.

Further reading

- Apple's Core Spotlight documentation: <https://developer.apple.com/documentation/corespotlight>
- Allowing Apps and Websites to Link to Your Content: https://developer.apple.com/documentation/uikit/core_app/allowing_apps_and_websites_to_link_to_your_content

26

Extending iMessage

As part of Apple's effort to introduce more and more extension points into the iOS ecosystem, iMessage extensions were added in iOS 10. iMessage extensions allow your users to access and share content from your app with their friends right inside the Messages app. This chapter will show you exactly how to create an iMessage extension.

You will learn about two different extension types: Sticker packs and iMessage apps. Note that these extensions are called apps and not just extensions. This is because iMessage extensions behave a lot like apps even though they're not apps. Also, iMessage apps are distributed through their own iMessage App Store. You will learn more about this later in this chapter.

First, you will gain a broad understanding of iMessage apps and what possibilities they give you as a developer. Once this is established, you will deep dive into several aspects of iMessage apps in more detail. To be more specific, the following are the topics that will be covered in this chapter:

- Creating a sticker pack for iMessage
- Implementing custom, interactive iMessage apps
- Understanding sessions, messages, and conversations

By the end of this chapter, you'll understand how iMessage apps work and how you can create an experience for your users that is fun and engaging. More importantly, you'll be able to launch an amazing sticker pack that features cats, dogs, or any other type of sticker you'd like to slap on messages you send and receive.

Understanding iMessage apps

An iMessage app is really an app extension that lives inside iMessage. It's kind of a weird type of extension because it behaves like a *hybrid between an extension and an application*. Extensions typically contain a single view controller and, more importantly, they can't be distributed through the iOS ecosystem without containing an application.

This rule does not apply to iMessage apps. An iMessage app can be distributed directly through the iMessage app store without a containing application. This means that you can build an iMessage app, such as a sticker pack, and distribute it without associating any other app to it. Also, you can add in-app purchases to an iMessage app. In-app purchases are normally unavailable to extensions, which makes iMessage apps behave more like an app than an extension.

Apart from distribution and in-app purchases, iMessage apps behave just like other extensions. They are created on behalf of a host application, Messages in this case, and they have a relatively short lifespan. An iMessage app specializes in sending messages from one user to another. These messages are shaped and created inside your extension and added to the conversation by sending them. An iMessage app can contain or send three different types of messages:

- Interactive messages
- Stickers
- Media content

An interactive message is intended to be tapped by the receiving user so they can interact with it somehow. Stickers are a special kind of image that the user can place right on top of other messages or send as a single image. Media content is distributed in the form of photos, videos, links, and any other type of content that iMessage natively supports. iMessage apps are built using the Messages framework. This framework has been created explicitly for iMessage extensions, and it functions as a gateway from your app into the conversation that your user is currently having.

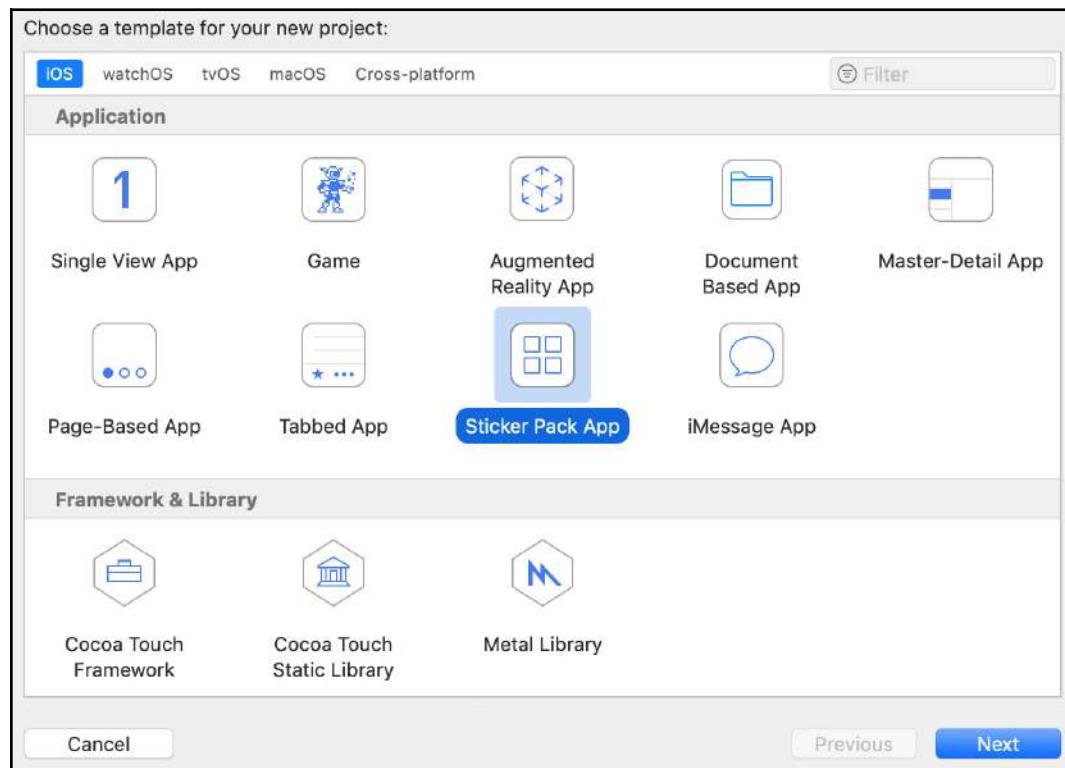
As mentioned before, iMessage apps can be distributed without a host application. This means that you can distribute your iMessage app independently of any other app. Even though app icons inside of images are not squares, you must still provide an app icon that is square and would otherwise be used on the user's home screen by a containing app. This image is required to display the extension in the settings app and other places in iOS where a square app icon is required.

If you do have a containing app for your iMessage app, users can directly add the iMessage app to their list of available apps through the iMessage App Store without having to download it separately. The simplest app you can build for iMessage is a sticker pack. Let's go ahead and create one so we can start exploring iMessage apps in the simulator.

Creating an iMessage sticker pack

Stickers in iMessage are a fun way to share images with your friends. Stickers can be slapped onto a message as a response to a message you've received or just for fun with no particular reason.

To create a sticker pack, open Xcode and create a new project. One of the available project templates is the **Sticker Pack App**. Select this template and click **Next**. Give your sticker pack a name and click **Next** again to create your project:



In the generated project, you'll find two folders. One is named `Stickers.xcstickers` and the other is named `Products`. When you create a sticker app, you're only interested in the `stickers` folder. If you open it, you'll find an app icon template and a folder named `Sticker Pack`.

All you need to do to create your sticker pack is to drag images into this `Sticker Pack` folder. After doing this, you can build and run your extension and try it out in the simulator. You'll find that the simulator comes with two active conversations in the Messages app. These are dummy conversations and you'll use them to test your iMessage app.

The simulator cannot send messages outside the simulator so you must use these predefined conversations while testing. Tapping a sticker or dragging it onto an existing speech bubble will send it from one conversation to the other. Go ahead and give that a shot. You can also send text messages and any other type of message you would normally send in the dummy conversation.

Optimizing assets for your stickers

Creating a sticker pack is really easy, and to make the best sticker packs, you should make sure that you properly optimize your image assets. Stickers can be displayed to the user in three sizes: small, medium, and large. To change the size of stickers in your sticker pack, simply click the `Sticker Pack` folder and open the attributes inspector. You can pick the sticker size there.

You should optimize the size of the stickers you provide according to the display size you choose. The following is a list of sticker sizes and their recommended asset sizes:

- **Small:** 100x100 @3x
- **Medium:** 136x136 @3x
- **Large:** 206x206 @3x

Note that all sizes have an @3x annotation. This means that the exported dimension of your images should be three times larger than the size listed to accommodate screens with a 3x resolution, such as the plus-sized iPhones. So, a small sticker should be exported at 300x300 pixels. Unlike other image assets on iOS, you only supply a single image for each sticker, the 3x image. Normally you supply a 1x, 2x, and 3x version of each asset you include in your app.

Sticker packs can contain PNG images, animated PNG images, GIFs, and JPEGs. All images you use must be smaller than 500 kb. It's recommended that you provide all of your assets as either PNG or animated PNG because this image format has superior image quality and supports transparency. Transparent stickers are recommended because they look a lot better when placed on top of other stickers or messages.

Now that you're familiar with sticker packs and testing them in the simulator, let's see how we make our sticker pack a bit more special by customizing the sticker-picking interface a bit.

Creating a custom sticker app

The standard sticker app that you created earlier using the sticker pack template is very plain. The generated iMessage app shows stickers on a white background and that's about it. Sticker apps shouldn't be more complex than this, but wouldn't it be great if you could at least change the background color for a sticker app? This isn't possible if you're using the simple sticker pack template. You can, however, create your own sticker pack app and customize the background. Doing this will allow you to familiarize yourself with the code that's involved in creating an iMessage app, so let's create a custom sticker pack.

In the source code repository for this book, you'll find a project named `CustomStickers`. This project already contains the stickers that will be used in the sample app. The images for these stickers are made available through openclipart.org by a user named bocian. Note that the images have not been added to the `Assets.xcassets` folder, but to the `Stickers` folder in `MessagesExtension`. The project was set up like this since the `Assets.xcassets` folder belongs to the containing app, which will be ignored.

In the `MessagesExtension` folder, you'll find a view controller file and a storyboard file. Open the storyboard and remove the default `UILabel` that was placed in the interface. You won't be adding any interface elements through Interface Builder because the interface elements to create a sticker app aren't directly available in Interface Builder.

You will find several boilerplate methods in the `MessagesViewController` class. We'll get into them soon; you can ignore them for now. The `viewDidLoad` method will be used to set up `MSStickerBrowserViewController` to display stickers in.

The `MSStickerBrowserView` instance that is contained inside `MSStickerBrowserViewController` behaves somewhat like `UICollectionView` because it requires a data source to determine how many, and which, stickers to display.

The first step in implementing your own sticker app is to add a property for an instance of `MSStickerBrowserViewController` to `MessagesViewController`:

```
var stickerBrowser = MSStickerBrowserViewController(stickerSize:  
    .regular)
```

Next, add the following implementation for `viewDidLoad`:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    stickerBrowser.willMove(toParent: self)  
    addChild(stickerBrowser)  
    stickerBrowser.didMove(toParent: self)  
  
    view.addSubview(stickerBrowser.view)  
    stickerBrowser.stickerBrowserView.dataSource = self  
    stickerBrowser.stickerBrowserView.reloadData()  
    stickerBrowser.stickerBrowserView.backgroundColor = UIColor.red  
}
```

The preceding snippet should not contain any surprises for you. First, `stickerBrowser` is added as a child view controller of the messages view controller. Then the `stickerBrowser` view is added as a subview of the messages view controller's view. Next, the `dataSource` is set on `stickerBrowserView` and it is told to reload its data. Lastly, the background color for `stickerBrowserView` is set.

If you build your app now, Xcode will complain about `MessagesViewController` not conforming to `MSStickerBrowserViewDataSource`. Add the following extension to `MessagesViewController.swift` to make `MessagesViewController` conform to `MSStickerBrowserViewDataSource`:

```
extension MessagesViewController: MSStickerBrowserViewDataSource {  
    func stickerBrowserView(_ stickerBrowserView: MSStickerBrowserView,  
    stickerAt index: Int) →; MSSticker {  
        return OwlStickerFactory.sticker(forIndex: index)  
    }  
  
    func numberOfStickers(in stickerBrowserView: MSStickerBrowserView) →  
    Int {  
        return OwlStickerFactory.numberOfStickers
```

```
    }  
}
```

The first method is expected to return a sticker for a certain index and the second returns the number of stickers in the app. The logic for this is abstracted into a sticker factory. This is done to keep the code in the view controller nice, compact, and to the point. Add a new Swift file to the project and name it `OwlStickerFactory`.

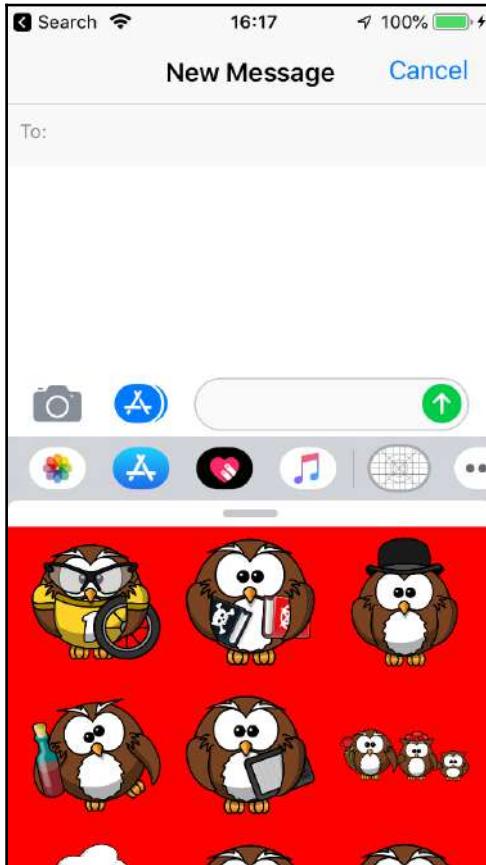
Add the following implementation to this file:

```
import Foundation  
import Messages  
  
struct OwlStickerFactory {  
    static private let stickerNames = [  
        "bike", "books", "bowler", "drunk", "ebook",  
        "family", "grill", "normal", "notebook", "party",  
        "punk", "rose", "santa", "spring"  
    ]  
  
    static var numberOfStickers: Int { return stickerNames.count }  
  
    static func sticker(forIndex index: Int) → MSSticker {  
        let stickerName = stickerNames[index]  
  
        guard let stickerPath = Bundle.main.path(forResource: stickerName,  
ofType: "png")  
            else { fatalError("Missing sticker with name: \(stickerName)") }  
        let stickerUrl = URL(fileURLWithPath: stickerPath)  
  
        guard let sticker = try? MSSticker(contentsOfFileURL: stickerUrl,  
localizedDescription: "\(stickerName) owl")  
            else { fatalError("Failed to retrieve sticker: \(stickerName)") }  
        return sticker  
    }  
}
```

Most of the preceding code should speak for itself. There is an array of sticker names and a computed variable that returns the number of stickers for the app. The interesting part of this code is the `sticker(forIndex:)` method. This method retrieves a sticker name from our array of names. Then it retrieves the file path that can be used to retrieve the image file from the application bundle. Finally, it creates a URL with this path to create a sticker.

Note that the `MSSticker` initializer can throw errors so the initialization call is prefixed with `try?`. Also, note that the sticker initializer takes `localizedDescription`. This description is used by screen readers to read out a description of the sticker to users that have certain accessibility features enabled.

When you run your extension now, you should see an interface that looks very similar to the following screenshot:



You can drag stickers from the sticker browser to the messages view. Notice that the stickers are picked and placed with a nice peel animation. Whenever you add an `MSSticker` to the messages view, regardless of whether you use a sticker browser, the Messages framework takes care of the sticker peel, drag, and drop animations for you. This means that you can create a custom interface for your stickers if you'd like.

Keep in mind that most sticker apps will make use of the standard layout and your users might not be too pleased if your app presents them with an unexpected sticker sheet layout. However, apps that aren't about stickers do require a special layer of design and interaction. This is the next topic covered in this chapter.

Implementing custom, interactive iMessage apps

Sticker apps are nice, but they're not particularly interactive. You can build far more interesting and interactive applications for iMessage through the **Messages** framework. Some of the larger, well-known apps on iOS have been able to implement iMessage extensions that make sharing content from their apps easier. There are even people that have built games in iMessage! The **Messages** framework allows developers to build a wide range of extensions straight into the **Messages** app.

You have just seen how you can build sticker packs and how you can create a somewhat customized sticker pack by picking the app template instead of the sticker pack template in Xcode. You haven't gone in-depth into the different life cycle methods that Xcode generates for you when you create a new iMessage app.

To go into more depth with the **Messages** framework, you will build an iMessage app for **The Daily Quote**. You built a widget for this app in [Chapter 23, Instant Information with a Today Extension](#). First, you will learn what the life cycle for an iMessage app looks like. After that, you will implement your own iMessage app.

Understanding the iMessage app life cycle

An iMessage app lives inside the **Messages** app, just like you would expect from an extension. As mentioned before, iMessage apps are a special kind of extension, which makes them behave a lot like apps even though they are still extensions at their core.

The main view controller for an iMessage app must always be a subclass of `MSMessagesAppViewController`. You can't have a primary view controller that is not a subclass of this class. When the user navigates to your extension through the iMessage apps drawer, this view controller is added where the keyboard would normally be placed. This size for the extension is called compact mode.

When the Messages framework instantiates your extension, the `willBecomeActive(with:)` method is called, followed by `didBecomeActive(with:)`. These methods are called after `viewDidLoad()` in the view controller life cycle, but before `viewWillAppear(_:)`. When it's time to dismiss your extension, the `viewWillDisappear(_:)` and `viewDidDisappear(_:)` life cycle methods are called. Next, `willResignActive(with:)` and `didResignActive(with:)` are called.

Once the resignation methods are called, the process for your iMessage app is killed shortly thereafter. You do not get any time to do work in the background. This is also true for other extension types. Again, even though Messages extensions behave a lot like apps, they're not.

There are two more methods you should know about for now. These methods are called whenever your iMessage app is transitioning from one display mode to another. The display mode is changed whenever the user expands the iMessage app by dragging the handle that is shown above your extension's view.

You can also trigger this transition from code by calling `requestPresentationStyle(_:)`. The life cycle methods that get called when the extension changes its size are `willTransition(to:)` and `didTransition(to:)`. The first method is called right before the transition occurs, the second is called right after.

Implementing the custom compact view

To start implementing the Messages extension for **The Daily Quote**, a new extension should be added to the project. Pick the **Messages extension** and name it **The Daily Quote Messages**. Enable the **App Groups** capability for this extension and include `Quotes.swift` in the extension target, just like you've done before.

You will use the same view-controller containment technique you used when building the custom sticker pack. This time you will add the view controller that should be displayed to the extension's Storyboard without connecting it to `MessagesViewController`. Before you do this and create the interface, you should implement the code for the compact view. The compact view will feature a quote, the creator of the quote, and a **Share** button.

Create a new `UIViewController` subclass and name it `CompactViewController`. Make sure to add it to the correct target by selecting the extension's folder before creating the new file. The setup for this view controller will be really similar to all of the other view controllers you've created for **The Daily Quote** and its extensions. The view will contain two outlets, one for the quote and one for its creator. In `viewDidLoad()`, the current quote should be fetched and the labels should be populated with the correct values. I trust you to be able to do this on your own. When in doubt, check the other projects. Alternatively, check the source code in the code bundle for this chapter.

When a user taps the Share button, the current quote should be shared as a message in the active conversation. This means that `MessagesViewController` will have to be aware of the share command that occurs in `CompactViewController`. To do this, you can create a `QuoteSelectionDelegate` protocol to which `MessagesViewController` can conform.

The delegate and the share action should be implemented in `CompactViewController`, as shown in the following code snippet:

```
var delegate: QuoteSelectionDelegate?  
  
@IBAction func shareTapped() {  
    delegate?.shareQuote(Quote.current)  
}
```

The delegate is optional because it can't be set before the view controller is initialized. The tap action simply calls a method on the delegate and passes the current quote along with it. Create a new file to define the protocol and name it `QuoteSelectionDelegate`. The protocol should be implemented as follows:

```
protocol QuoteSelectionDelegate {  
    func shareQuote(_ quote: Quote)  
}
```

This is a simple protocol with just a single method requirement. Now, let's write all the required code in `MessagesViewController` before implementing the interface for `CompactViewController`. First of all, add the property shown in the following snippet to the message view controller so it can hold on to the compact view controller. Also, update your `viewDidLoad` implementation as follows:

```
var compactViewController: CompactViewController?  
  
override func viewDidLoad() {  
    super.viewDidLoad()
```

```
compactViewController =
storyboard?.instantiateViewController(withIdentifier:
"CompactViewController") as? CompactViewController
compactViewController?.delegate = self
}
```

The `instantiateViewController(withIdentifier:)` method from `UIStoryboard` is used to obtain an instance of `CompactViewController` from the storyboard.

Add an extension to `MessagesViewController` so it conforms to `QuoteSelectionDelegate`. Add an empty implementation for `shareQuote(_:)` to the extension. You will implement the share functionality later.

Because the user can switch between the compact and expanded state as they please, it's a good idea to abstract the code to show the compact and expanded views into their own methods. Doing this will make your code easier to read, and it avoids code duplication. Add the following method to `MessagesViewController` to show the compact view controller:

```
func showCompactView() {
    guard let compactViewController = self.compactViewController
    else { return }

    compactViewController.willMove(toParent: self)
    addChild(compactViewController)
    compactViewController.didMove(toParent: self)

    view.addSubview(compactViewController.view)
    compactViewController.view.frame = view.frame
}
```

This code is very similar to what you saw when you implemented the sticker pack. The last step to show the compact view is to implement `willBecomeActive(with:)`. `showCompactView()` will be called from this method as follows:

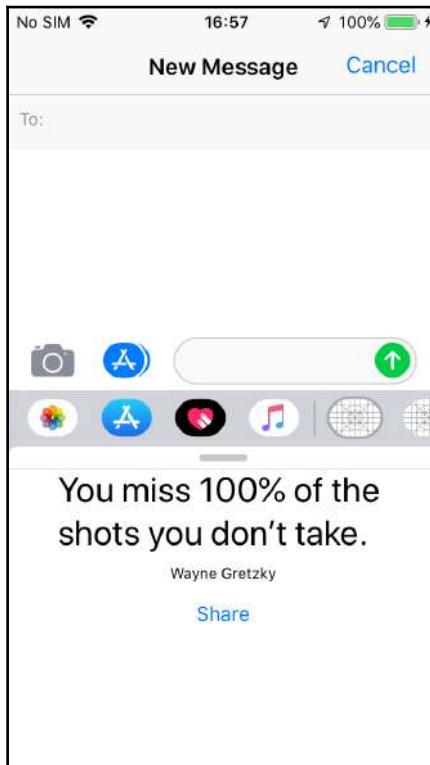
```
override func willBecomeActive(with conversation: MSConversation) {
    if self.presentationStyle == .compact {
        showCompactView()
    }
}
```

Now, open `MainInterface.storyboard` and drag out a view controller. Add two labels and a button to this view controller. Lay them out as shown as shown in the following screenshot:



The quote is styled as a title and its maximum number of lines is set to 0 lines, so it automatically expands to fit the content. The creator is styled as caption one, and it's positioned below the title. The button should be laid out below the caption. Make sure to set `CompactViewController` as both the subclass and the storyboard ID for the view controller you just added. To wrap it up, connect the outlets to the views. Make sure to select **Touch up inside** as the trigger action for `shareTapped()`.

If you build and run your application now, the quote for today should pop up right inside iMessage:



This is all you need to do to implement the compact view. Now let's implement an expanded view for the iMessage app.

Implementing the expanded view

The expanded view for the iMessage app will be a table view that lists all of the quotes in the `Quote` struct. You will use a similar setup to the one you used before by creating a new view controller file and using the delegate protocol to communicate the selection of a quote back to `MessagesViewController`.

First, create a new `UITableViewController` subclass and name it `QuotesTableViewController`. You can remove most of the commented template code; the only methods you should keep are `tableView(_:didSelectRowAt:)`, `tableView(_:numberOfRowsInSection:)`, and `numberOfSections(in:)`. In addition to the commented delegate methods, you can remove the `viewDidLoad()` and `didReceiveMemoryWarning()` methods; you don't need them.

For starters, you should implement the methods shown in the following code snippet. These methods provide the table view with its data:

```
override func numberOfSections(in tableView: UITableView) → Int {
    return 1
}

override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) → Int {
    return Quote.numberOfQuotes
}

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) → UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
    "QuoteTableViewCell", for: indexPath)
    let quote = Quote.quote(atIndex: indexPath.row)

    cell.textLabel?.text = quote?.text
    cell.detailTextLabel?.text = quote?.creator
    return cell
}
```

The preceding snippet uses a `numberOfQuotes` property on the `Quote` struct. However, this property is not defined yet. Add it to the `Quote` struct, as shown in the following code snippet:

```
static var numberOfQuotes: Int { return quotes.count }
```

The last thing you should take care of before creating and connecting the interface is the quote-selection delegate and implementing cell selection in `QuotesTableViewController`. The code to do this is pretty straightforward:

```
var delegate: QuoteSelectionDelegate?

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    guard let quote = Quote.quote(atIndex: indexPath.row)
        else { return }
    delegate?.shareQuote(quote)
}
```

This leaves you with a complete implementation of the table view in code. Let's create the interface in Interface Builder. Open `MainInterface.storyboard` and drag out a table view controller. Assign `QuotesTableViewController` as its class and Storyboard ID. Also, click the prototype cell and set its style to **Subtitle**. The Identifier for the cell should be set to `QuoteTableViewCell`. That's all you need to do for now. Let's make sure that you display this view controller when the iMessage app is in the expanded presentation mode.

In `MessagesViewController`, update `willBecomeActive(with:)` so it can display both the expanded and compact mode as the initial view, depending on `presentationStyle`:

```
override func willBecomeActive(with conversation: MSConversation) {
    if self.presentationStyle == .compact {
        showCompactView()
    } else if self.presentationStyle == .expanded {
        showExpandedView()
    }
}
```

Since the user can manually change the display mode for the iMessage app, you will need to dynamically switch the view controller that is shown depending on the presentation style. Add the following implementation for `willTransition(to:)`:

```
override func willTransition(to presentationStyle:  
MSMessagesAppPresentationStyle) {  
    if presentationStyle == .compact {  
        showCompactView()  
    } else if presentationStyle == .expanded {  
        showExpandedView()  
    }  
}
```

Finally, you should take care of showing the correct view. You should always make sure to remove any existing view controllers before you show a new one. `showExpandedView()` has not been implemented yet, but if you think about what this method should do for a second, you will realize it should look very similar to the `showCompactView()` method. It's a good idea to implement a special cleanup method and refactor `showCompactView()`, so you can reuse it for `showExpandedView()`. First, add the following method that will be used to clean up the displayed view controllers when needed:

```
func cleanupChildViewControllers() {  
    for viewController in children {  
        viewController.willMove(toParent: nil)  
        viewController.removeFromParent()  
        viewController.didMove(toParent: nil)  
  
        viewController.view.removeFromSuperview()  
    }  
}
```

One last thing that needs to be done before implementing the presentation of the expanded view controller is adding a property for this expanded view controller in `MessagesViewController` and updating `viewDidLoad()`, so it creates a new instance of `QuotesTableViewController` that will be used as the expanded view controller. Add the following property and the `viewDidLoad()` implementation to `MessagesViewController`:

```
var expandedViewController: QuotesTableViewController?  
  
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    compactViewController =
```

```
storyboard?.instantiateViewController(withIdentifier:  
    "CompactViewController") as? CompactViewController  
    compactViewController?.delegate = self  
  
    expandedViewController =  
storyboard?.instantiateViewController(withIdentifier:  
    "QuotesTableViewController") as? QuotesTableViewController  
    expandedViewController?.delegate = self  
}
```

Next, let's implement the refactored view controller presentation methods:

```
func showCompactView() {  
    guard let compactViewController = self.compactViewController  
    else { return }  
  
    showViewController(compactViewController)  
}  
  
func showExpandedView() {  
    guard let expandedViewController = self.expandedViewController  
    else { return }  
  
    showViewController(expandedViewController)  
}  
  
func showViewController(_ viewController: UIViewController) {  
    cleanupChildViewControllers()  
  
    viewController.willMove(toParent: self)  
    addChild(viewController)  
    viewController.didMove(toParent: self)  
  
    view.addSubview(viewController.view)  
    viewController.view.frame = view.frame  
}
```

After doing this, build and run the app to see your extension in action. It flawlessly switches between showing the list of quotes and the single daily quote.

The final step in implementing our iMessage app is to implement the `shareQuote(_:)` method. This method will compose a message that can be shared. Let's have a look at message composition and everything related to it.

Understanding sessions, messages, and conversations

This chapter has mostly focused on user interface-related aspects of iMessage apps. When you create an iMessage app, your app will sooner or later have to send a message. To do this, you use the `MSMessage` and `MSSession` objects. In addition to messages and sessions, the `MSConversation` class represents the conversation context that your extension exists in. These three classes together allow you to send messages, identify recipients in conversations, and even update or collapse existing messages in the messages transcript.

When an iMessage extension is activated, `willBecomeActive(with:)` is called in `MessagesViewController`. This method receives an instance of `MSConversation` that you can use to send messages, stickers, and even attachments. More importantly, the conversation contains unique identifiers for participants in the conversation and the currently selected message.

The `localParticipantIdentifier` and `remoteParticipantIdentifier` properties identify the current user of your app or the person who is sending messages with your app, and the recipients of these messages, respectively. Note that these identifiers are unique to your app so you can't use this identifier to identify users across different iMessage apps. Also, these identifiers can change if the user uninstalls and then reinstalls your app. The participant identifiers are mostly meant to identify users in a group conversation.

The `selectedMessage` property is only set if your extension was launched due to a user tapping on a message that was sent by your extension. This is especially useful if you're building an app that sends multiple messages that build upon each other. In Apple's 2016 WWDC talk, *iMessage Apps and Stickers, Part 2*, they demonstrated an ice-cream designer app. Each user designs a part of the ice cream and `selectedMessage` is used in combination with `MSSession` to collapse old messages so the message history does not get polluted with old pictures of ice creams. You will learn how to use this session in a bit. First, let's have a look at `MSMessage` to see how you can create messages.

Composing a message

The `MSMessage` class encapsulates all the information that is contained inside a message. When you initialize a message, you can initialize it with or without a session. If you instantiate a message with a session, other messages that are attached to this session will be collapsed to only show the message's `summaryText`. The summary is supposed to be a short, clear description of the message so that, even when the message is collapsed, the message still makes sense.

If possible, you should always aim to attach a `URL` and `accessibilityLabel` to your messages, depending on the types of message your app will send. If you're sending plain text messages like **The Daily Quote** will, you don't need an accessibility label since iOS can simply read the text message out loud for users with accessibility needs. The `URL` property is mainly used by platforms that don't support iMessage extensions, such as macOS. Attaching a `URL` makes sure that users can still navigate to your content online. When you're sending plain text, this isn't an issue.

Finally, if you're sending a message that has media attached to it, you should make use of the `MSMessageTemplateLayout` class. `MSMessage` has a `layout` property that can be set to an instance of `MSMessageTemplateLayout`. The message layout template is highly configurable. You can assign an image or media to it, set a caption, an image title, a sub-caption, and more. Messages will make sure that your layout looks good and is laid out nicely depending on the information you set.

If you set both a media URL and an image on your message layout, the image will be used and the other media items are ignored. Any images you add should be 300x300 @3x. Avoid rendering text on the image as the scaling on different devices might degrade the quality of your image and render the text illegible. Instead, use the image title and image subtitle properties from the message layout object.



An `MSMessage` instance is always intended to either have some form of media associated with it or to be interactive.

Sending a message

Once you have composed a message, you need to attach it to a conversation. To do so, you can use the `activeConversation` property, which is already present on your `MSMessagesAppViewController`. There are several methods on `MSConversation` that will insert a message. You can insert an instance of `MSMessage` if you have composed one. Alternatively, you can insert a sticker or you can simply insert some text.

To share a quote with **The Daily Quote** messages extension, this means that the text from the selected quote should be inserted into the conversation. Add the following implementation for `shareQuote(_:)` to implement the share feature:

```
func shareQuote(_ quote: Quote) {
    guard let conversation = activeConversation
    else { return }

    conversation.insertText("(quote.text) - (quote.creator)",
    completionHandler: nil)
}
```

If you run this code and try sharing a quote, two noteworthy things occur:

- The extensions presentation mode does not change to compact after sharing a quote so the user doesn't see that the quote is shared.
- When you change the app to compact mode manually, the quote isn't actually sent yet. Instead, a message is prepared for the user to send manually. Users are always in control of when a certain message is sent. Extensions are only allowed to prepare messages for users so they can decide for themselves whether the message should be sent.

To fix the problem with the extension not switching to compact mode after selecting a quote, you need to add a single line of code to `shareQuote(_:)`. After inserting the quote, add the following line:

```
dismiss()
```

This will dismiss the extension entirely, allowing the user to focus on the quote and send it to the recipient. Alternatively, you could call `requestPresentationStyle(_:)` with a `compact` presentation style to change the current presentation style for your messages extension.

Once the user decides that they want to send the message you've composed on their behalf, they must manually send the message. Once this happens, `didStartSending(_:conversation:)` is called on your messages view controller. You won't be notified when this message was successfully sent to the recipient. This means that you can't make any assumptions in your app about whether the message was delivered to the recipient.

If the user decided to not send your `MSMessage` instance, `didCancelSending(_:)` is called. You can use this method to clean up or do something else in response to the cancel event. Note that both the start and cancel methods are only called when you're working with `MSMessage` instances. If you're sending a simple text, as **The Daily Quote** does, you won't be notified about these events at all.

Summary

In this chapter, you learned everything about iMessage apps. A messages extension is very similar to an app, and it's an exciting, powerful way for you to integrate your apps further into the iOS ecosystem. You now know how you can create a sticker pack to provide your users with a fun, refreshing way to share imagery from your app through standard and custom sticker packs.

Most importantly, you learned that even though iMessage apps can be very complex, they can also be simple. Some iMessage apps will make heavy use of interactive messages, others are merely standard sticker packs, or somewhere in between, such as **The Daily Quote**. The possibilities are endless and you can have a lot of fun with iMessage extensions. Just be aware that during testing you'll only be able to use your extension in the simulator so, unfortunately, you can't surprise your friends by suddenly sending them stickers from a sticker pack you're still developing.

This chapter wraps up the advanced integrations section of the book. The following chapters will focus on ensuring your app is of top-notch quality and submitting it to the App Store.

Questions

1. In what way are iMessage Extensions similar to apps?
 - a) They can exist independently and support In-App Purchases.
 - b) They can use view controllers.
 - c) They are not similar to apps at all.
2. What do you need to create a Sticker Pack?
 - a) You need images that can be added to the Sticker Pack template.
 - b) You need a simple extension that provides stickers to the sticker browser.
 - c) You need an extension that implements the sticker's peel and drop animations.
3. How can you test your Message extensions?
 - a) You can use your extensions on a device and send content to your friends.
 - b) You must use the predefined conversations in the simulator.
 - c) Both a) and b).
4. What object is used to configure a complex message that can be sent through your extension?
 - a) The `MSMessageContent` object.
 - b) The `MSConversationMedia` object.
 - c) The `MSMessage` object.
5. How can you identify the users that participate in a conversation?
 - a) By reading their unique, random identifiers.
 - b) By retrieving their iCloud accounts.
 - c) By their name and account details.
6. How do you know whether a message was delivered to the recipients?
 - a) `didStartSending(_:conversation:)` is called on the extension.
 - b) `didFinishSending(_:conversation:)` is called on the extension.
 - c) There is no way to be sure due to privacy concerns.

Further reading

- **iMessage Apps and Stickers, Part 2:** <https://developer.apple.com/videos/play/wwdc2016/224/>
- **Apple's Messages framework documentation:** <https://developer.apple.com/documentation/messages>

27

Discovering Bottlenecks with Instruments

To properly debug and improve your apps, you need to understand what tools are available to you. One of the tools Apple ships as part of Xcode is called **Instruments**. **Instruments** contains a collection of measurement tools that help you to profile and analyze your app to debug and detect complex problems or performance bottlenecks. For example, Instruments can help you figure out whether your app is suffering from memory leaks. Tracking a memory leak without the right tools is tedious and nearly impossible. A tool such as Instruments can help you track down several possible causes for a memory leak, which can save you a lot of time.

In this chapter, we're going to look at an app named **Mosaic**. This app is a straightforward app that was built to demonstrate how you can profile your app. The app is very slow; the longer it's used and the more the user interacts with the app, the slower the app becomes. You will learn how you can use Instruments to find the problems in **Mosaic** so you can eventually fix them, so the app runs smoothly.

To wrap this chapter up, you'll learn about a feature that is brand new in Xcode 10: Custom Instruments. With **Custom Instruments**, you can add your own measurements and visualizations to Instruments. This enables you to build tailor-made profiling tools that are perfect for your own app or framework.

This chapter is divided into the following segments:

- Exploring the Instruments suite
- Discovering slow code
- Closing memory leaks
- Building your own Instruments

By the end of this chapter, you'll be able to analyze your apps to find potential issues and solve them before they become a problem.

Exploring the Instruments suite

In this book's code bundle, you'll find a project named **Mosaic**. The app is still in the early stages of development, so there is plenty of work that needs to be done to improve it. More specifically, the app doesn't seem to work very well.

Even though the app isn't finished and the code isn't perfect, the app *should* be working fine. None of the features appears to be very performance-heavy, so the app works fine. If you launch the app and click around, you'll notice that the app isn't working fine. It's really slow! Let's see how you can track down the exact cause of the slowness.

If you dig through the source code, you'll immediately see that the app has three main screens:

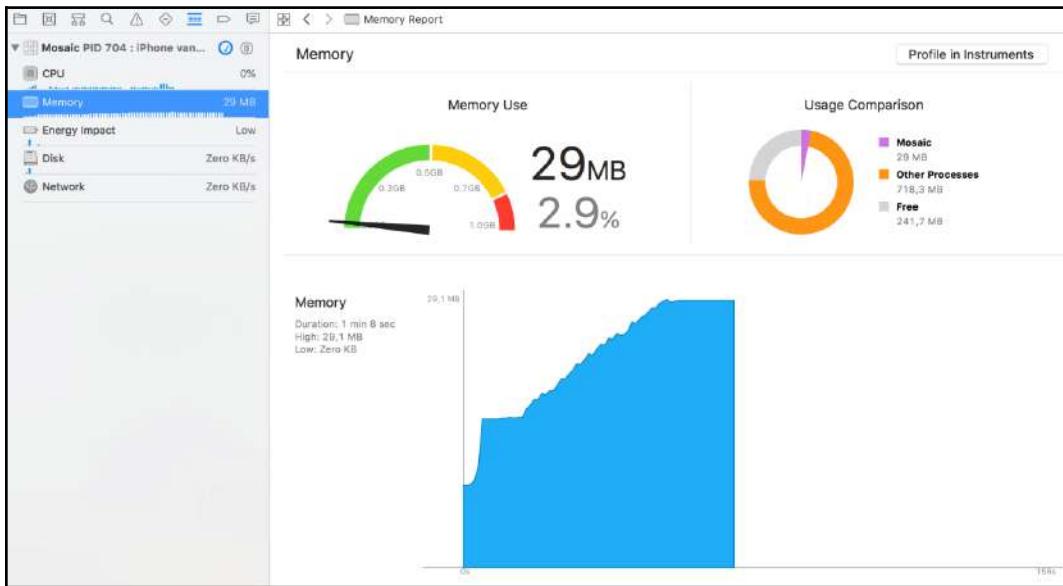
- A table view
- A collection view
- A detail view

The table view displays a list of 50 items, each of which links to an infinitely-scrolling collection view with a custom layout. The layout for the collection view is custom because it needed to have exactly one pixel of spacing between items. The detail view only shows an image, nothing too crazy.

Based on knowledge from earlier chapters, you should know that a simple app such as **Mosaic** should be able to run smoothly on virtually any iOS device. Collection views and table views are optimized to display huge datasets, so an endlessly-scrolling collection view that keeps adding 50 new rows when you almost reach the end of the list should not be too much of a problem. In other words, this app should be able to run at 60 frames per second with no trouble at all.

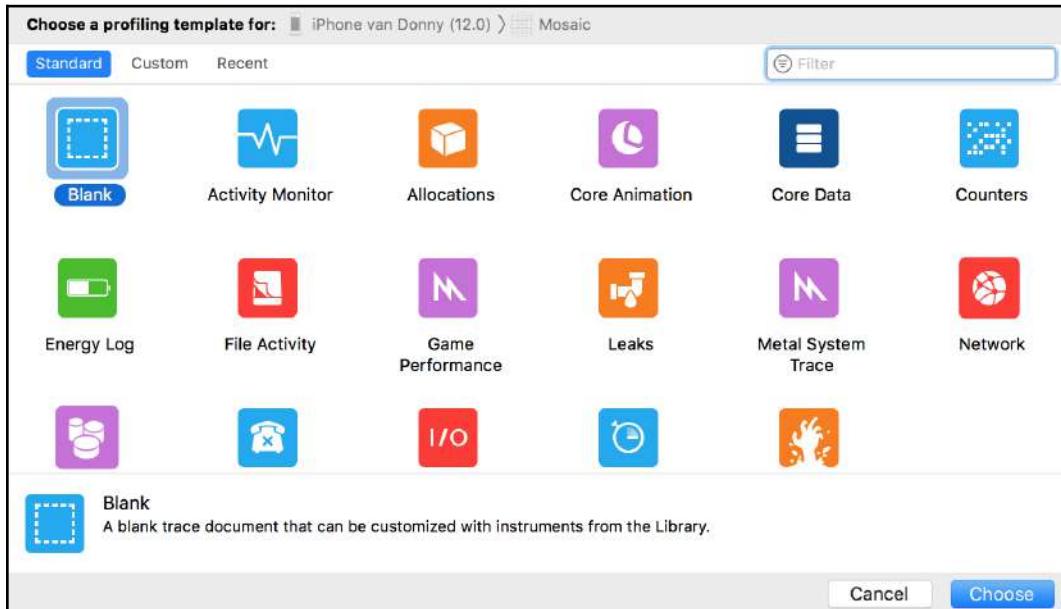
However, if you start scrolling through the collection view and move back and forth between screens, and then scroll more and more, you'll find that some collections will stop scrolling smoothly. They will randomly freeze for a little while and then continue scrolling. And the more you scroll through the collections, the worse this issue seems to become.

If you take a look at the memory usage in Xcode's Debug navigator (the sixth icon in the left sidebar), you can tell that something is wrong because memory usage keeps going up while navigating through screens. This in itself is nothing to worry about, temporary memory spikes are normal. When the used memory never gets freed even though you might have expected it to, that's when you should start worrying. Take a look at the following graph:



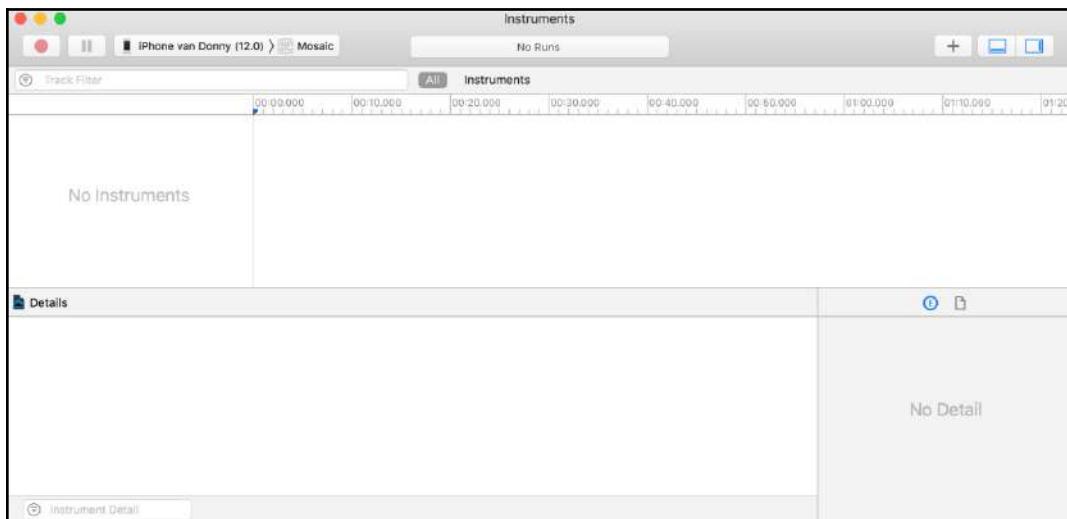
Whenever you debug performance issues or if you're trying to measure performance, make sure to measure on a real device. If you measure using the simulator, your app might run fine even though it might not run fine on a device. This is due to the way the simulator works. It mimics an iOS device, but it's backed up by all the power your Mac has to offer. So, the processor and available memory are those of your Mac, resulting in performance that's often much better than it would be on a device.

For now, make a mental note of the app's rising memory usage. Let's have a look at the Instruments app. To profile your app with Instruments, you can either select **Product | Profile** from the Xcode's toolbar or hit *Cmd + I* on your keyboard. Xcode will then build your project and launch Instruments so you can select a profiling template:



There are many available templates, and most of them are created for a specific case. For instance, the **Network** template will measure network usage. There are also templates available to profile animations, database operations, layout, Metal graphics performance, and more.

If you select a blank template, you're presented with the Instruments interface. The interface contains a record button that is used to start or stop profiling for an app. If you look further to the right side of the top toolbar, there's a plus icon. You can use this icon to add profiling tools for specific metrics yourself. You could use this to create a profiling suite that fits your needs without having to switch between the default templates all of the time. The center of the Instruments interface is the timeline. This is where graphs are drawn, representing the measurements that were made for your app. The bottom section of the screen shows detail about the currently selected metric, as you're about to find out:



Now that you know a little bit about the interface of the Instruments app, let's make an attempt at profiling the **Mosaic** app. The biggest issue in the app is the choppy scrolling when you scroll down the collection view for several pages. It's clear that the app has a memory issue, as you saw in Xcode, but choppy scrolling occurs even if the app's memory usage is low. This should be a flag that a memory leak probably isn't the problem that is causing the collection views to scroll badly. Something in the code is probably slow, and Instruments can help to figure out which parts of the app's code are taking a long time to complete.

Discovering slow code

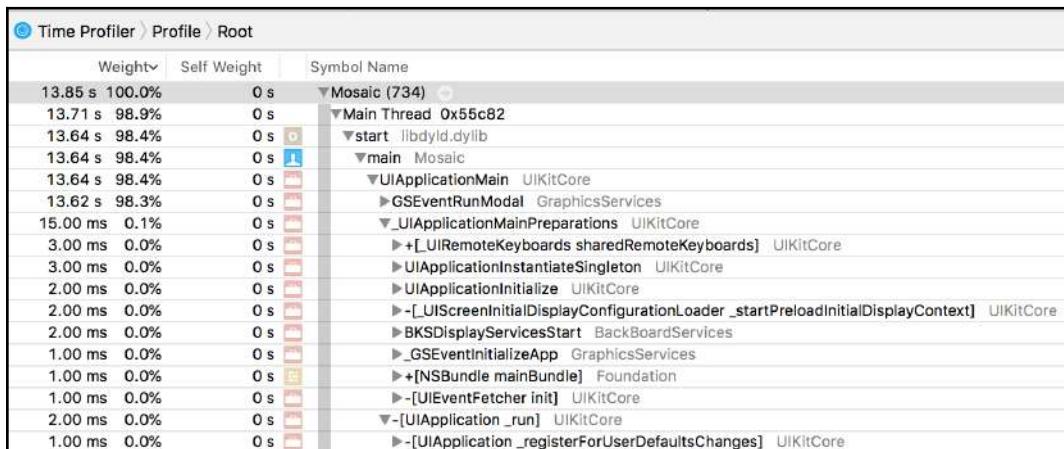
Whenever you find that your app is slow or choppy, there's a good chance that something in your code is taking longer than it should, especially if your memory usage appears to be within reasonable range. For instance, if your app uses less than 50 MB, memory is not likely to be an issue for your app, so seeking the problem in your code's performance makes a lot of sense.

To discover where your code is lacking in the performance department, it's a great idea to profile your app in Instruments, by either selecting **Product | Profile** in Xcode's toolbar or by pressing *Cmd + I*. To figure out what the code is doing, you need to select the **Time Profiler** template when Instruments asks you which template you want to use. This template measures how long certain blocks of code take to run.

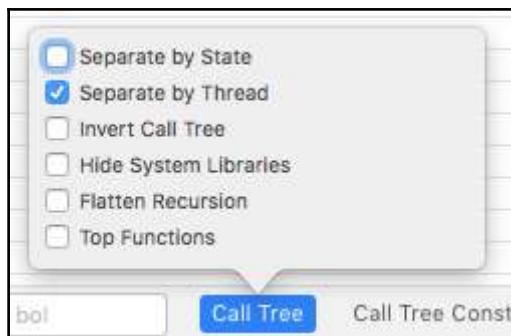
To record a profiling session of our app, make sure that a device is connected to your Mac and make sure that it's selected as the device that your app will run on by selecting your iOS device from the list of devices and simulators in the scheme toolbar menu in Xcode. Once you've selected your device, start profiling the app. When Instruments launches, pick the **Time Profiler** template and hit record. Now use the app to navigate to a collection and begin scrolling until the app starts acting choppy and scroll some more. After seeing the app stutter a couple of times, there should be enough data to begin filtering out what's going on. Press the stop button to stop the recording session.

If you take a look at the data Instruments has recorded, you'll notice a graph that has a bunch of peaks. This part of the timeline is marked CPU, and if you hit *Cmd + a* couple of times to zoom in on this timeline, you'll notice that these spikes seem to last longer and longer as the scrolling becomes choppier and choppier. This is a great clue to investigate a bit more. Something must be going on that makes these peaks last longer every time they occur.

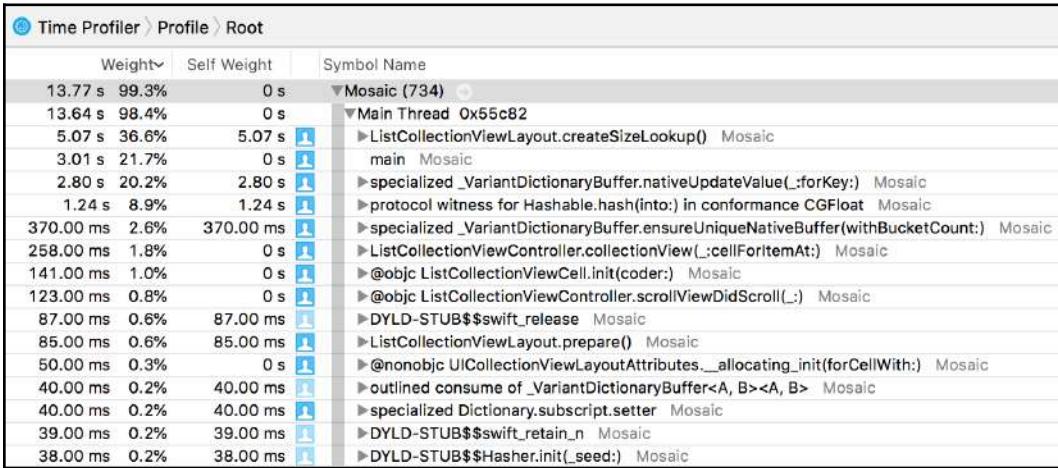
In the bottom section of the Instruments window, you'll find an overview of the code in the app that was executed while the recording session was active. The code is separated by thread and since you saw the user interface lagging, you can be pretty sure that something on the main thread is slow. If you drill down into the list of method calls by a couple of levels, you won't find much useful information. It doesn't even look like most of the executed code is part of the app's code, as you can see in the following screenshot:



To make sure the code you're interested in is pulled to the surface, you should apply a couple of filtering options. If you click on the **Call Tree** button in the bottom of the window, you can select **Invert Call Tree** and **Hide System Libraries**:



The **Invert Call Tree** option inverts the tree of method calls. The default tree starts at `main` and works its way down to the method calls in your app code. When you invert this tree, your app code will be at the top and works its way down to `main`. By checking **Hide System Libraries**, you make sure that all code shown in the detail view belongs to your app. The call tree is a lot more readable after applying this filter:



When you examine the call tree, it's immediately clear that the issue is in the collection view layout. On the left side of the detail area, the percentage of time spent in a certain method is outlined, and you can see that a considerable portion of time is spent in `ListCollectionViewLayout.createSizeLookup()`. This method appears to be much slower than it should be.

It's probably tempting to immediately point at the `createSizeLookup()` method and call it slow. It would be hard to blame you if you did, that method is a pretty rough one because it's computing an entire layout. However, the layout should be computed pretty efficiently. Unfortunately, you can't prove this with the data in Instruments. If you could see the time that each method call takes, you would be able to verify the claim that `createSizeLookup()` is not slow.

What can be proved, though, is that `createSizeLookup()` is called way more often than it should be. If you add a `print` statement at the start of this method, you'll see that there are hundreds of prints as you scroll through the list. If you dig deeper and figure out when `createSizeLookup()` is called exactly, you'll find two places:

- In `prepare()`
- In `rectForThumbAtIndex(_:)`

This is strange because `rectForThumbAtIndex(_:)` is called in a loop inside of the `prepare` method after the size lookup has already been generated.

More importantly, as the number of items in the collection grows, the number of items to loop over grows too. It's very likely that this extraneous call to `createSizeLookup()` is the source of the bug. You should be able to remove the call to `createSizeLookup()` from `rectForThumbAtIndex(_:)` without problems because the lookup is already created before the items are iterated, so it's safe to assume that the lookup exists when `rectForThumbsAtIndex(_:)` is called. Remove the call to `createSizeLookup()` and run your app on the device. Make sure to do plenty of scrolling to verify that the bug was fixed.

To make sure that everything is fixed, you should use Instruments again to see what the Time Profiler report looks like now. Start a new Time Profiler session in Instruments and repeat the steps you performed before. Navigate to a collection and scroll down for a while so the app loads lots of new pages.

Looking at the result of the measurements now, the percentage of time spent in `createSizeLookup()` has dramatically decreased. It has decreased so much that it's not even visible as one of the heaviest methods anymore. The app performs way better now and you have acquired the measurements to prove it.

Now that the scrolling performance issue is solved, you can navigate the app, and everything is working smoothly. But if you take a look at the memory usage in Xcode, the memory usage still goes up every time you navigate to a new item in the table view. This means that the app probably has another problem in the form of a memory leak. Let's see how to discover and fix this leak with Instruments and Xcode.

Closing memory leaks

Usually, if you navigate around in your app, it's normal to see memory usage spike a little. More view controllers on a navigation controller's stack mean that more memory will be consumed by your app. This makes sense. When you navigate back, popping the current view controller off the navigation controller's stack, you would expect the view controller to be deallocated and the memory to be freed up.

The preceding scenario describes how **Mosaic** should work. It's OK if the app uses some more memory if you're deeper in the navigation stack, but this memory should be freed up after the back button is tapped.

In the **Mosaic** app, the memory keeps growing every time you navigate to a new collection view. It doesn't matter if you drill deep into the navigation stack, hit back, or scroll a lot, once memory is allocated it never seems to be deallocated. This is a problem, and you can use Instruments to dig into the app to find out what's wrong. Before you do this, though, let's have a deeper look at memory leaks, how they occur, and what the common causes are.

Understanding what a memory leak is

When your app contains a memory leak, this means that it's using more memory than it should. More specifically, the app fails to release memory that is no longer needed. A couple of scenarios exist where this could occur. Once you're aware of these scenarios, they should be quite easy to spot. If you haven't seen them before or have never even heard of them, it's easy to fall into the trap of having a memory leak in your app.

Preventing objects from using infinite memory

The first type of memory leak is one where an object is allowed to take up an endless amount of memory without any restrictions. A typical example of this is a cache. When you implement a cache that holds on to certain model objects, API responses, or other data that was expensive to obtain in the first place, it's easy to overlook the fact that you have just built yourself a memory leak.

If your user is using your app and your cache object keeps caching more and more data, the device will eventually run out of memory. This is a problem because if the app doesn't free the memory in time, it will be terminated by iOS to make sure that essential processes and other apps don't suffer because of your app's out-of-control memory usage.

Luckily, it's easy to solve issues such as these. The operating system will notify your app through `NotificationCenter` any time it needs you to free up memory.

Listening to this notification and purging any cached data you can recreate or reload will prevent your app from hogging memory and, ultimately, it prevents your app from being terminated due to memory reasons.

Here is a straightforward example of an image cache class that purges its cache when memory is tight:

```
class ImageCache: NSObject {
    var cache = [UIImage]()

    override init() {
        super.init()

        NotificationCenter.default.addObserver(self, selector:
#selector(purgeCache), name:
UIApplication.didReceiveMemoryWarningMemoryWarningNotification, object: nil)
    }

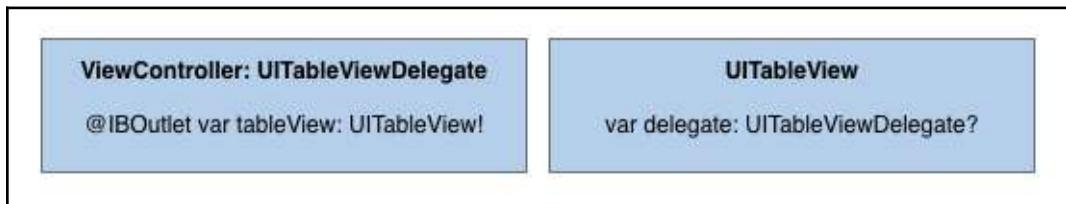
    deinit {
        NotificationCenter.default.removeObserver(self, name:
UIApplication.didReceiveMemoryWarningMemoryWarningNotification, object: nil)
    }

    @objc func purgeCache() {
        cache.removeAll()
    }
}
```

All you need to do is listen for the `UIApplication.didReceiveMemoryWarningMemoryWarningNotification` notification and purge any data that you can recreate when needed. Even though it's technically not required, this sample unsubscribes the cache from the memory warning notifications when it's deallocated. Since iOS 9, the OS itself should take care of this but this behavior has proven to be somewhat unpredictable so unsubscribing explicitly avoids potential reference cycles, which is the next type of memory leak to explore.

Avoiding reference cycles

When an object contains references to other objects, you should always be careful to avoid situations where both objects continuously hold a reference to each other. For example, a relationship between a table view and its delegate or data source could become a reference cycle if the relationship isn't managed correctly. Objects can only be deallocated if no objects are referencing them anymore:



The preceding figure illustrates this. The view controller holds onto `tableView` and `tableView` holds onto its delegate, which is the view controller. This means that neither object can ever be deallocated because for both the view controller and `tableView`, there is always at least one object referencing each at any given time. Of course, Apple has made sure that this doesn't occur in your apps by making sure that `tableView` does not hold onto its delegate forever. You'll see how in just a second.

Another situation where a reference cycle could be created is in a closure. When you implicitly reference `self` in a closure, the compiler complains that you must explicitly refer to `self`. Doing this creates a reference to `self` inside of the closure, potentially resulting in a reference cycle. Throughout this book, you've seen a bunch of closures and they always used a capture list `self` was used inside of the closure:

```
api.fetchData { [weak self] in
    self?.tableView.reloadData()
}
```

The preceding example shows an example of using a capture list. The capture list is the part right before the `in` keyword. The list captures a weak reference to `self`, which means that no reference cycle is created between the closure and `self`. If the `api` object stores the closure in a variable and you didn't use a weak reference to `self`, you might have a reference cycle. If the `api` object itself is held onto by another object, you can be pretty sure that a reference cycle is created.

Making the reference `weak` tells the app that the reference to `self` does not add up to the reference count of `self`. This means that if there are only weak references left to an object, it's OK to deallocate it and free the memory. Memory management and reference counts aren't simple. One way to think about this subject is that your app has an internal count of the number of objects that point to another object. For instance, if you create an instance of `UIView` inside of `UIViewController`, the reference count for `UIView` is one. When `UIViewController` is deallocated, the reference count for `UIView` becomes zero because the view controller doesn't use it anymore, meaning that it can be deallocated safely.

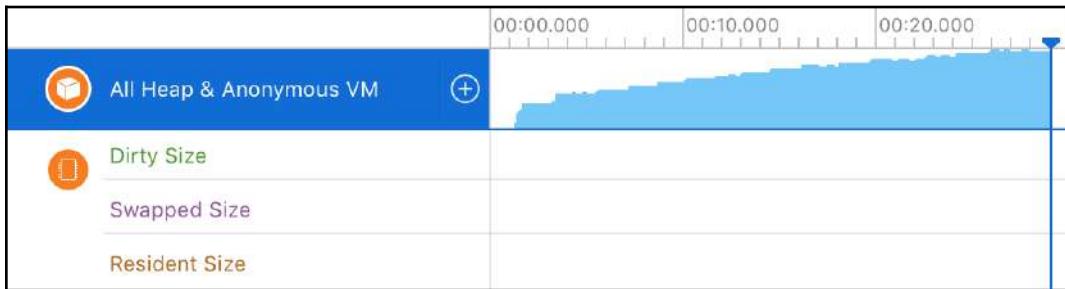
If `UIView` has a reference to `UIViewController` as well, both objects will keep each other around because the reference count for each instance won't ever reach zero. This is called a reference cycle. This cycle can be broken by marking at least one of the references as a `weak` reference. Since `weak` references don't contribute to the reference count, they prevent reference cycles from happening. This is how Apple has made sure that `tableView` does not create a reference cycle with its delegate or data source; the references to these objects are marked as `weak`.

As an alternative to making a reference `weak`, you can mark it as `unowned`. While `weak` is essentially a safe optional value, `unowned` makes the object implicitly unwrapped. It's often best to take the safe route and mark a captured reference as `weak` because your app won't crash if the `weak` referenced instance has been deallocated somehow, while it would crash if the reference is `unowned`.

Reference cycles aren't easy to understand, especially if you take into account `weak` references and reference-counting. It's really easy to get confused about this topic. Luckily, the **Mosaic** app contains a couple of issues with references and retain cycles so you can immediately put your newly acquired knowledge to the test.

Discovering memory leaks

To figure out why the memory usage of **Mosaic** increases every time a new screen is loaded in the app, you should profile the app using the Allocations Instruments template. When you've started a new Allocations profiling session, navigate through the app and you should see the memory usage graph rise consistently. This behavior is typical for a memory leak, so it's time to dig in deeper to figure out what exactly is going on that causes this to happen. Take a look at the following screenshot:



If you look at the detail area in Instruments, there is a lot of information there that does not make a lot of sense. A lot of the objects that are created are not objects that you explicitly wanted to create, which makes it hard to figure out what's causing the memory problem. It's not very likely to be an issue with UIKit, for instance, and even if it is, you can't quite fix that. Luckily, you can use the search dialog in the top-right corner of the detail area to look for objects that do belong to the application code, giving you insight into what's happening with the objects that get created by the application code.

If you type the word **Mosaic** in the search box, you'll find a couple of objects, and they should all look familiar to you. In the detail area, you can now see the number of instances of a particular object that is in memory. Refer to the following screenshot:

The screenshot shows the Instruments Statistics view with the 'Allocation Summary' section selected. At the top, there is a breadcrumb navigation: 'Allocations > Statistics > Allocation Summary'. Below the navigation, there is a table with the following columns: Graph, Category, Persistent Bytes, # Persistent, # Transient, Total Bytes, and # Total. The table lists five objects from the Mosaic application:

Graph	Category	Persistent Bytes	# Persistent	# Transient	Total Bytes	# Total
<input type="checkbox"/>	Mosaic.ListViewCell	134,53 KiB	210	0	134,53 KiB	210
<input type="checkbox"/>	Mosaic.ListViewController	12,69 KiB	14	0	12,69 KiB	14
<input type="checkbox"/>	Mosaic.ListViewLayout	4,59 KiB	14	0	4,59 KiB	14
<input type="checkbox"/>	Mosaic.AppDelegate	880 Bytes	1	0	880 Bytes	1
<input type="checkbox"/>	Mosaic.ListViewController	32 Bytes	1	0	32 Bytes	1

If you look closely, you'll find that there are way more collection view cells, collection view controllers, and collection view layouts present than you'd expect. The app can only show one collection view at a time, and when you navigate back to the list view controller, you would assume the collection view controller to be deallocated since nothing should be referencing it anymore.

When you segue to one of the collection view controllers, it's added to the `viewControllers` array on the navigation controller. This means that the navigation controller keeps a reference to each of the view controllers that are part of the navigation stack. This means that none of the view controllers in that list should be deallocated because the reference count is at least 1.

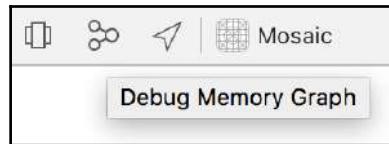
When you pop back to the list view controller, the navigation controller removes the collection view controller from its `viewControllers` array. The result of this is that the reference count for the collection view controller is now decremented by one since the navigation controller is not referencing it anymore. This should put the reference count at 0, meaning that the collection view controller can be deallocated and the memory should be freed up.

However, something is preventing this from happening, because the collection view controller remains allocated according to the observations in Instruments.

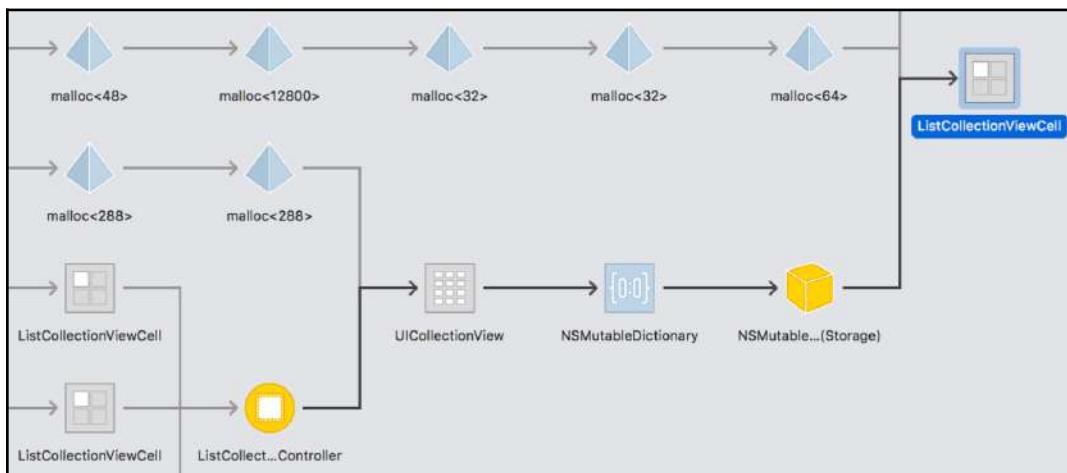
Unfortunately, Instruments does not tell you much more than the information just saw. Objects are sticking around for too long, which means that a reference cycle exists somewhere. This probably means that something is referencing the collection view controller and in turn, the collection view controller is referencing something else.

To figure out what's going on, you should probably start looking for problems in the collection view controller. The collection view controller has a delegate relationship with an object conforming to `ListCollectionDelegate`. It also acts as a delegate for the list collection view cells it displays, and it's also a delegate for the detail view. All the delegate relationships are references to other objects. These delegates could very well be causing the reference cycle. You can use Xcode to visualize all the objects in memory and see how they relate to each other. This means that it's possible to capture the state of the app's memory after browsing to several collection views, and you can see which objects are holding references to other objects. This enables you to visually identify reference cycles instead of blindly guessing.

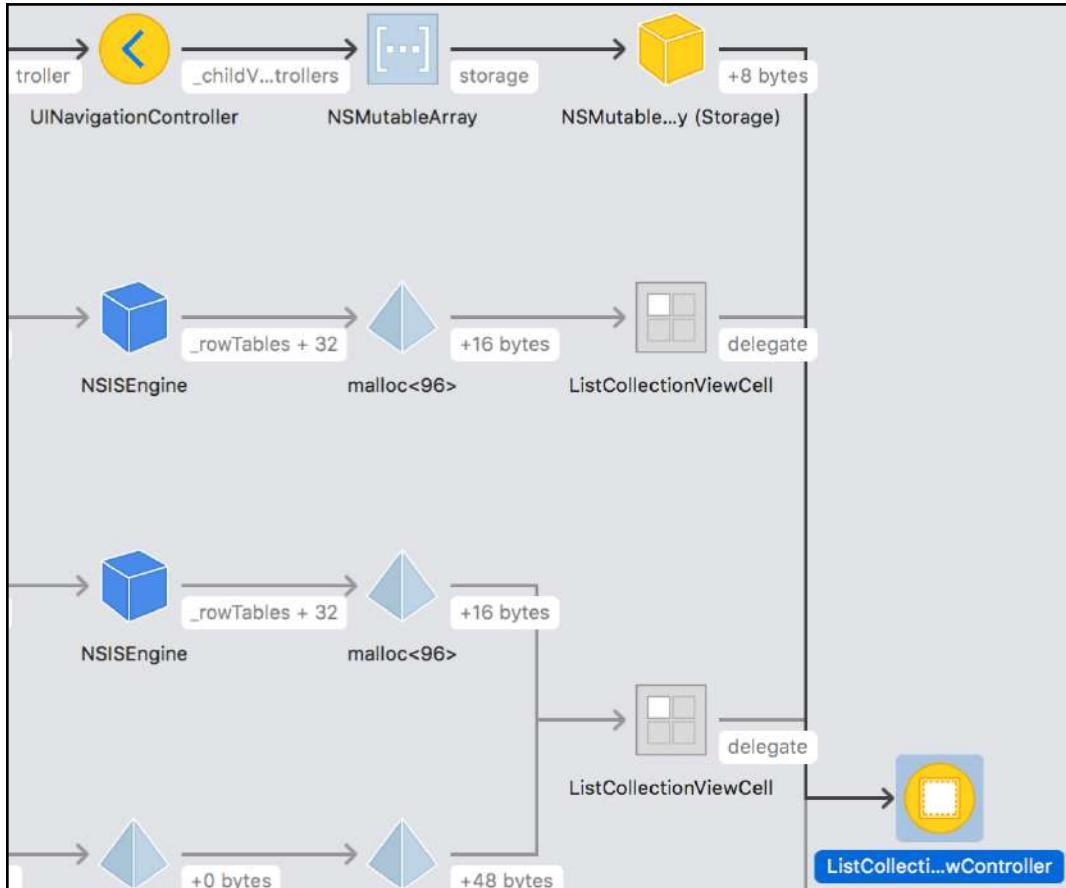
To visualize your app's memory usage, build and run the app in Xcode and navigate to a couple of collection views. Then open the memory view in the Debug navigator in Xcode and click the **Debug Memory Graph** button in the bottom toolbar of the screen:



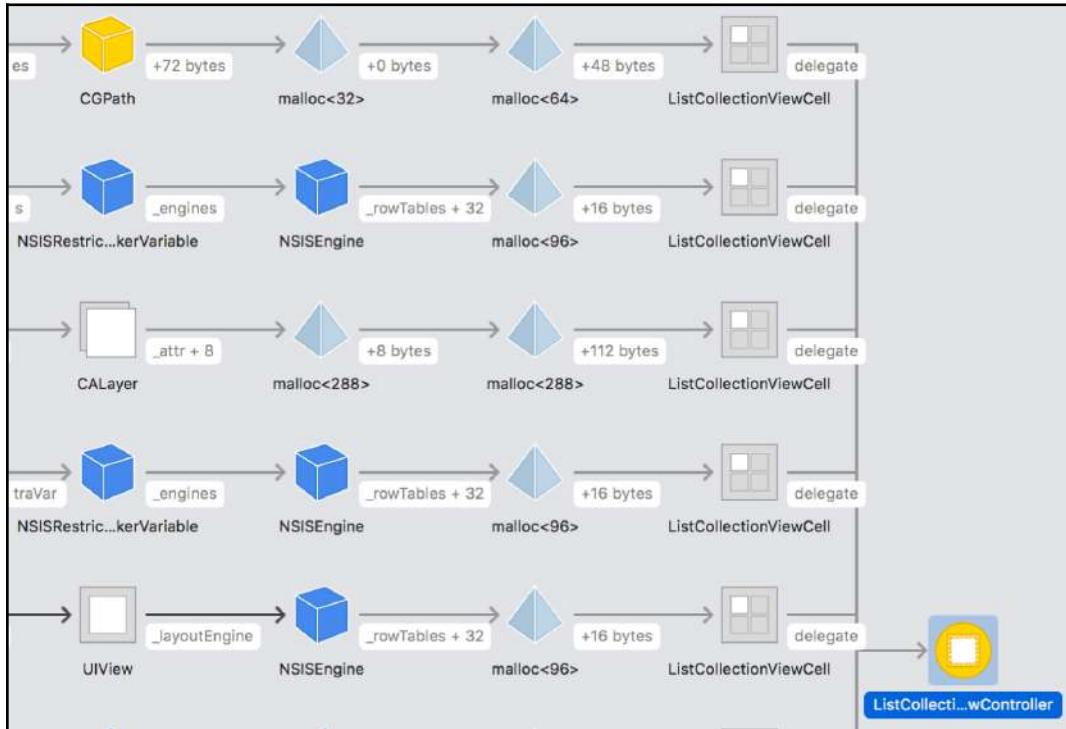
After clicking this button, Xcode will show you a visual representation of all the memory that your app is using. In the toolbar on the left side of the screen, look for `ListCollectionViewCell` and click it. Xcode will show you the relationship between `ListCollectionViewCell` and other objects. At first glance, nothing unusual is happening. There is a list view controller, which holds a reference to a collection view, then there are a couple of other objects, and finally there's the collection view cell:



Next, click the collection view controller. When you click it, two different views could be shown to you. One of these two shows a couple of collection view cells pointing to the collection view controller and there should be a navigation controller in the graph as well:



The other situation you might see is slightly different. It also has all the cells pointing to the collection view controller, but the navigation controller is nowhere to be found, as shown in the following screenshot:



This is pretty typical for a reference cycle. Objects that should be long gone stick around because somewhere, somehow, they are keeping references to each other. The memory graph reveals that the delegate relationship between the collection view controller and the collection view cells could very well be causing problems, you can begin fixing them.

In this case, the fix is pretty simple. You need to make the delegate relationship between the cell and the controller weak. Do this by opening `ListCollectionViewCell.swift` and changing the delegate declaration as shown:

```
weak var delegate: CollectionItemDelegate?
```

Marking the delegate as weak breaks the reference cycle because the references aren't strong anymore. This essentially tells our app that it's OK if the referenced object is deallocated. One final adjustment you must make is to constrain `CollectionItemDelegate` to class instances only. Due to the nature of value types, such as structs, you can't mark references to them as `weak`, and since protocols can be adopted by both value and reference types, the protocol must be constrained to class instances only. Update the declaration for the `CollectionItemDelegate` protocol as shown:

```
protocol CollectionItemDelegate: class
```

If you run the app now, you can safely navigate around and both the memory debugger and Instruments will prove that you no longer have any reference cycles in your app. It's possible that Instruments still tells you that your memory usage is growing. Don't worry about this too much; you did your job making sure that your code is safe and that all memory that you don't use anymore can be freed up.

The example of a reference cycle you just saw is a very common one. A common way to avoid reference cycles in delegates is to try to make all of your delegates weak by default. This prevents you from making mistakes that you might not catch right away and it's often the safest route to go. Do note that there are many ways in which a reference cycle can occur, and you should always use the appropriate tools to troubleshoot your memory issues.

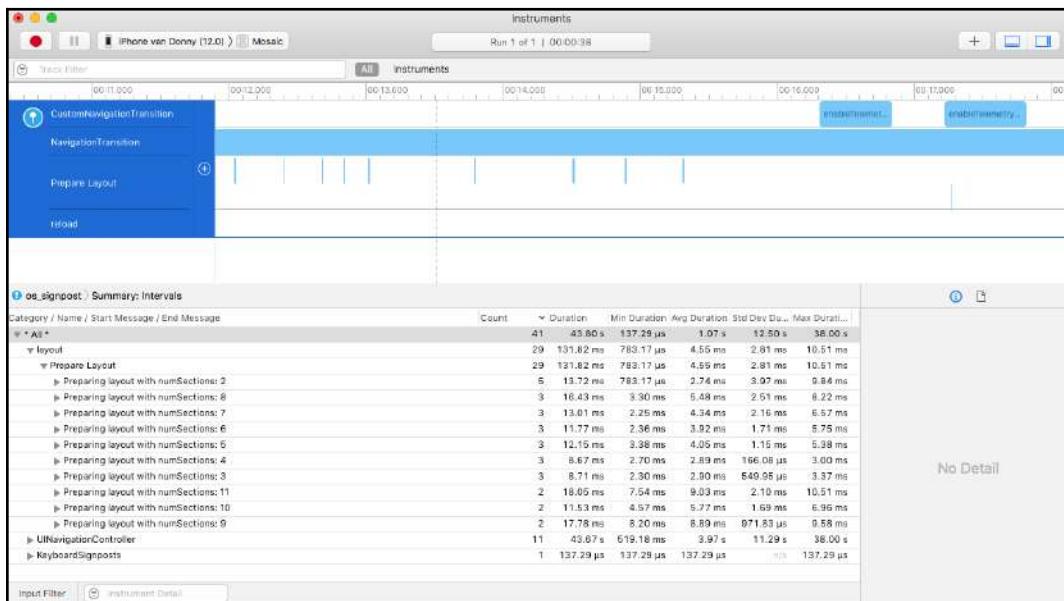
Let's wrap up this exploration of Instruments by creating an Instrument of your own.

Creating your own Instrument

The Instruments app comes with some great built-in tools that can be used to profile applications. While this is great, sometimes you might need something more tailor-made. Since Xcode 10, you can create your own Instruments Packages to profile your apps. Custom Instruments can use special logging that you add in your app through the `os_signpost` APIs to keep track of certain processes in your app while collecting interesting data about it. In this section, you will learn how you can add signpost logging to your app as a lightweight mechanism to measure performance in your app. Then, you'll learn how to take this data and funnel it through your own custom Instruments Package.

Adding signpost logging to your app

As a developer, you want to know exactly what your app is doing and how long your app spends on certain parts of your code. It's common to use breakpoints or print statements to do some basic debugging or logging in your app, but keeping track of many print statements in the console can be quite tedious. In iOS 12, you can use signpost logging to flag the beginning and end of a certain task and you can visualize these logs in Instruments using the built-in **os_signpost instrument**. The osSignpost instrument logs all signposts you track and shows them on a timeline. The following image shows an example of what signpost logging in Instruments looks like:



You can see a couple of timelines, one of them is labeled **Prepare Layout**. This timeline is a custom signpost that is sent from the **Mosaic** app. In the detail area, you can see more details about the signposts that were posted. The signposts that are posted by **Mosaic** correspond with the calls to `prepare()` on the custom collection view layout on the collection view page. Using signposts to track the performance of the `prepare()` method is more convenient in certain ways than using the Time Profiler; the Time Profiler provides a lot of detailed information that you might not always want to see right away. In this case, you would be more interested in making sure setting up the layout doesn't take too long. Additionally, you might want to make sure that `prepare()` performs well with many different amounts of sections.

Adding signposts to your app only takes a couple of minutes and they can give you fantastic new insights into what your app is doing. To add signposts to your app, you need an `OSLog` handle. An `OSLog` handle sounds very fancy, but it's just an instance of `OSLog` that is associated with your app. Go ahead and open `ListCollectionViewLayout.swift` in the **Mosaic** project and add the following code to this file:

```
import os.signpost

private let layoutLogger = OSLog(subsystem: "com.donnywals.layout",
category: "layout")
```

The preceding code imports the `os.signpost` framework and creates a log handle that will be used to track the signposts. When you create a signpost, you can flag the signpost as one of three types:

- Begin
- End
- Point of interest

The **begin** and **end** signposts are used to flag the start and end of an operation. You will add these signposts to the `prepare()` method to track the performance of preparing the collection view layout. A **point of interest** signpost is a special kind of signpost that is used to flag interesting moments that can occur in your app. For instance, when a user taps on something or when you reach a certain special moment inside of an operation. To log a signpost, you need to obtain `OSSignpostID`. This object is used to link two begin and end signposts together so it's essential that you use the same signpost ID. To create a signpost ID, you can use one of the following two methods:

```
let one = OSSignpostID(log: layoutLogger, object: self)
let two = OSSignpostID(log: layoutLogger)
```

The first method can only be used if the `object` parameter you pass to the `OSSignpostID` initializer is a class. This way of creating a signpost ID will always return the same for any given object. So if you use this method, every time you use this initializer inside of a particular class instance, the ID will be the same.

The second method generates a random, unique signpost ID every time. So if you're using signposts with a struct, make sure to store the signpost ID so you can reuse it if needed.

Once you have obtained your signpost ID, you can begin logging signpost events. Update the code for `prepare()` in the collection view layout as follows:

```
override func prepare() {
    super.prepare()

    guard let numSections = collectionView?.numberOfSections,
    numSections != 0
        else { return }
    // 1
    let id = OSSignpostID(log: layoutLogger, object: self)
    // 2
    os_signpost(.begin, log: layoutLogger, name: "Prepare Layout",
    signpostID: id, "Preparing layout with numSections: %{public}@",
    "\n(numSections)")

    // existing implementation

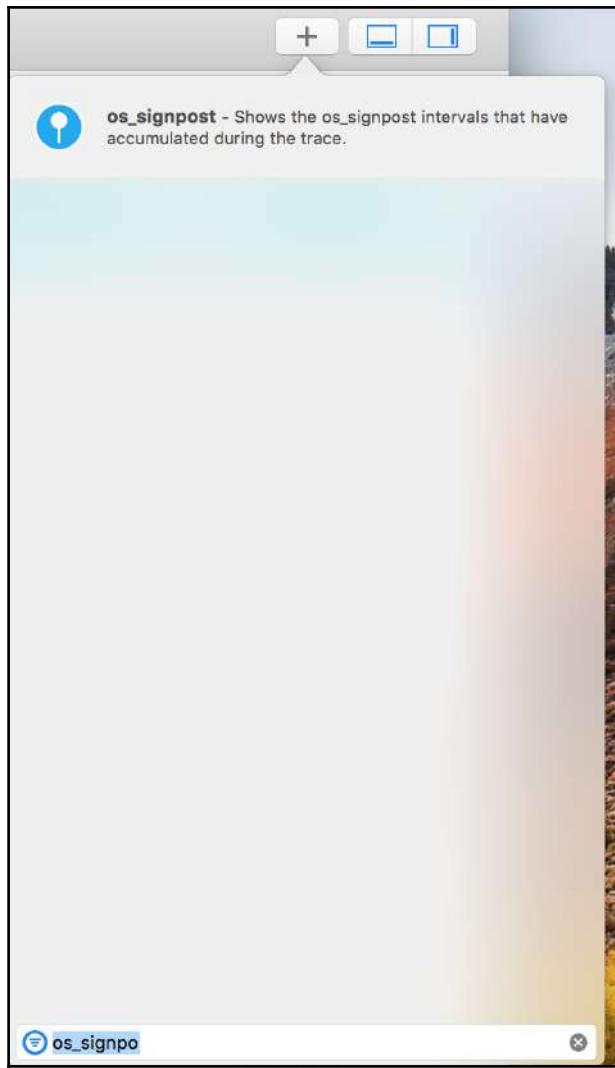
    // 3
    os_signpost(.end, log: layoutLogger, name: "Prepare Layout",
    signpostID: id, "Done preparing layout")
}
```

The first comment signals the creation of a signpost ID. Since the collection view layout is a class, it can be passed to the signpost ID initializer. Once the signpost ID is created, the begin signpost is added. Every signpost is attached to a logger and has a name. The name should be the same for the begin and end signpost so the system understands that they both relate to the same event in your app. The last argument for the signpost begin and end calls is the description of the event. The description is a formatted string so you can add metadata to it through format specifiers. In this case, the `%{public}@` specifier is used to signal that a string will be added to the description. The end signpost does not have any variables associated with it.



If you want to log a point-of-interest signpost, you log it similarly to how the begin and end signposts are logged, except you need a special kind of logger. To create a point-of-interest logger, you need to use `OSLog.Category.pointsOfInterest` as the category for the `OSLog` handle and then associate your signpost with that logger rather than your normal logger.

To see your signposts in Instruments, you need to run your app for profiling like you have done before in this chapter. Instead of selecting one of the predefined templates, select a blank template when Instruments starts up. Use the + icon to find the `os_signpost` instrument:

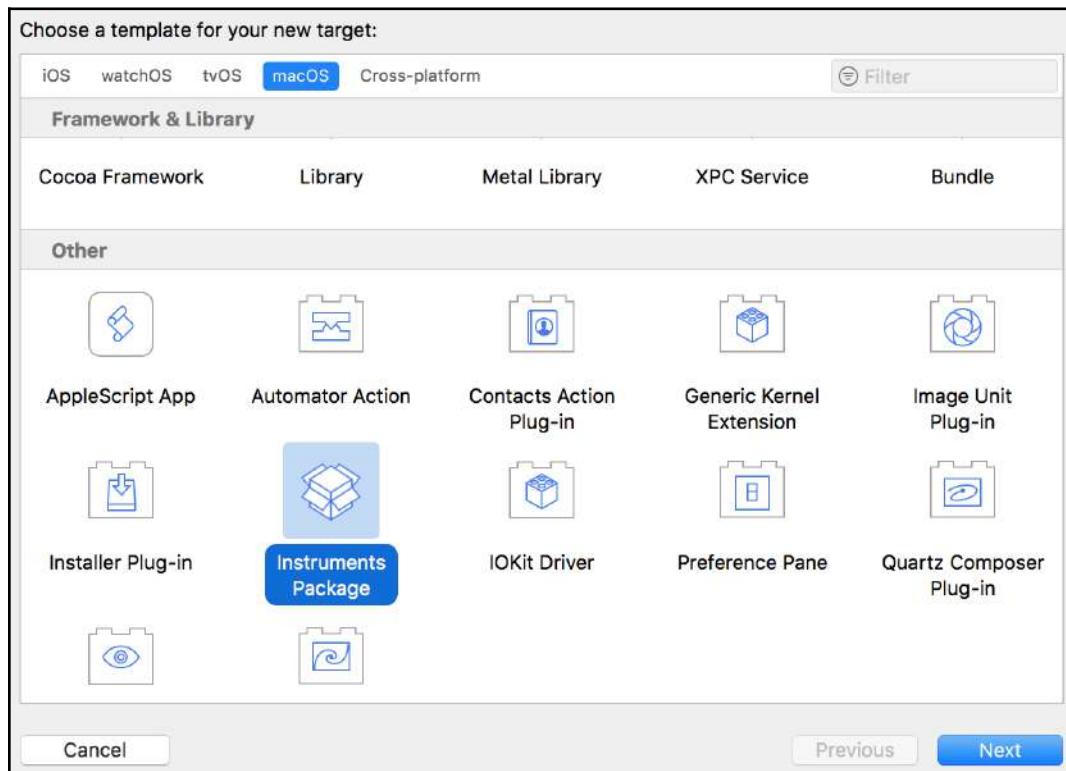


After locating the **os_signpost** instrument, drag it to the timeline area. You can now begin profiling your app and your signposts should appear. While this is cool, the default view combines your signposts in the detail view. You can see that every time you log *Preparing layout with numSections: 2*, the count for this message is incremented and you can see the minimum, maximum, and average duration. Usually, this is plenty of information, but what if you want to see every individual signpost call in the detail view? You can achieve this by creating your own custom Instruments Package.

Building an Instruments Package

Together with signpost logging, Apple introduced the ability to define custom Instruments Package. An Instruments Package exists as an XML file that describes the type of data that your app wants to track and how to visualize it. Apple uses this approach internally as well, which means that your own Instruments Packages can build upon Apple's own instruments packages. The most interesting type of custom instrument you can create is a signpost instrument. This type of instrument takes the data you log with the `os_signpost` API and you can decide how Instruments should visualize this data.

To create an Instruments Package, add a new target to your app. Instead of selecting the type of target from the list of iOS targets, go to the macOS tab and find the Instruments Package template:



Name the instrument **MosaicLayoutInstrument** if you want to follow along with the steps to create your own instrument. Xcode will create a new folder in your project that has a single file in it. The extension for this file is `instrpkg`. If you open this file, you will find a little bit of XML and a lot of commented-out information to guide you in creating your own instrument. You can remove all of these comments if you want.

Every instrument has roughly three main sections in the XML:

- Metadata about the instrument
- A schema that defines the data that is tracked
- The instrument itself

Depending on your exact needs, you might have multiple schemas or instruments. To track signposts, you need to add an `<os-signpost-interval-schema>` tag to the XML file. If you type `os-signpost-interval-schema` in the XML and hit *Esc*, Xcode will usually offer the option to automatically add a skeleton implementation for the implementation of the tag. The schema for this instrument will only track a single column, the number of sections that were involved in the layout calculation. Add the following implementation for the signpost interval schema:

```
<os-signpost-interval-schema>
  <!-- 1 -->
  <id>signpost-schema</id>
  <title>Layout Signpost Schema</title>

  <!-- 2 -->
  <subsystem>"com.donnywals.layout"</subsystem>
  <category>"layout"</category>
  <name>"Prepare Layout"</name>

  <!-- 3 -->
  <start-pattern>
    <message>"Preparing layout with numSections: "?num-
sections</message>
  </start-pattern>

  <end-pattern>
    <message>"Done preparing layout"</message>
  </end-pattern>

  <!-- 4 -->
  <column>
    <mnemonic>num-sections</mnemonic>
    <title>Number of Sections</title>
    <type>string</type>
```

```
<expression>?num-sections</expression>
</column>
</os-signpost-interval-schema>
```

The preceding snippet is defined in four sections:

1. Some metadata about the schema. The ID will be used later to refer to the schema in the instrument.
2. Information about how the information is logged. These fields should match the corresponding values in your app.
3. Patterns for the begin and end messages as they were used in your app. You use the `?num-sections` notation to indicate the location of a variable and to extract it into a new variable that can be used in your schema.
4. The columns that your schema contains. In this case, it's only a single column that uses `num-sections` as its identifier and corresponds with the `?num-sections` variable extracted from the begin message.

After adding the schema, you are ready to create the instrument. Add the following implementation to the XML file, after the schema definition:

```
<instrument>
  <!-- 1 -->
  <id>com.donnywals.layout-instrument</id>
  <title>Layout Instrument</title>
  <category>Behavior</category>
  <purpose>Insight in layout preparation</purpose>
  <icon>Generic</icon>

  <!-- 2 -->
  <create-table>
    <id>layout-table</id>
    <schema-ref>signpost-schema</schema-ref>
  </create-table>

  <!-- 3 -->
  <graph>
    <title>Layout preparation</title>
    <lane>
      <title>Prepare</title>
      <table-ref>layout-table</table-ref>

      <plot-template>
        <instance-by>num-sections</instance-by>
        <value-from>duration</value-from>
      </plot-template>
    </lane>
  </graph>
</instrument>
```

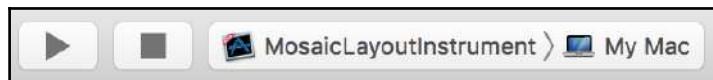
```
</graph>

<!-- 4 -->
<list>
    <title>Layout list</title>
    <table-ref>layout-table</table-ref>
    <column>num-sections</column>
    <column>duration</column>
</list>
</instrument>
```

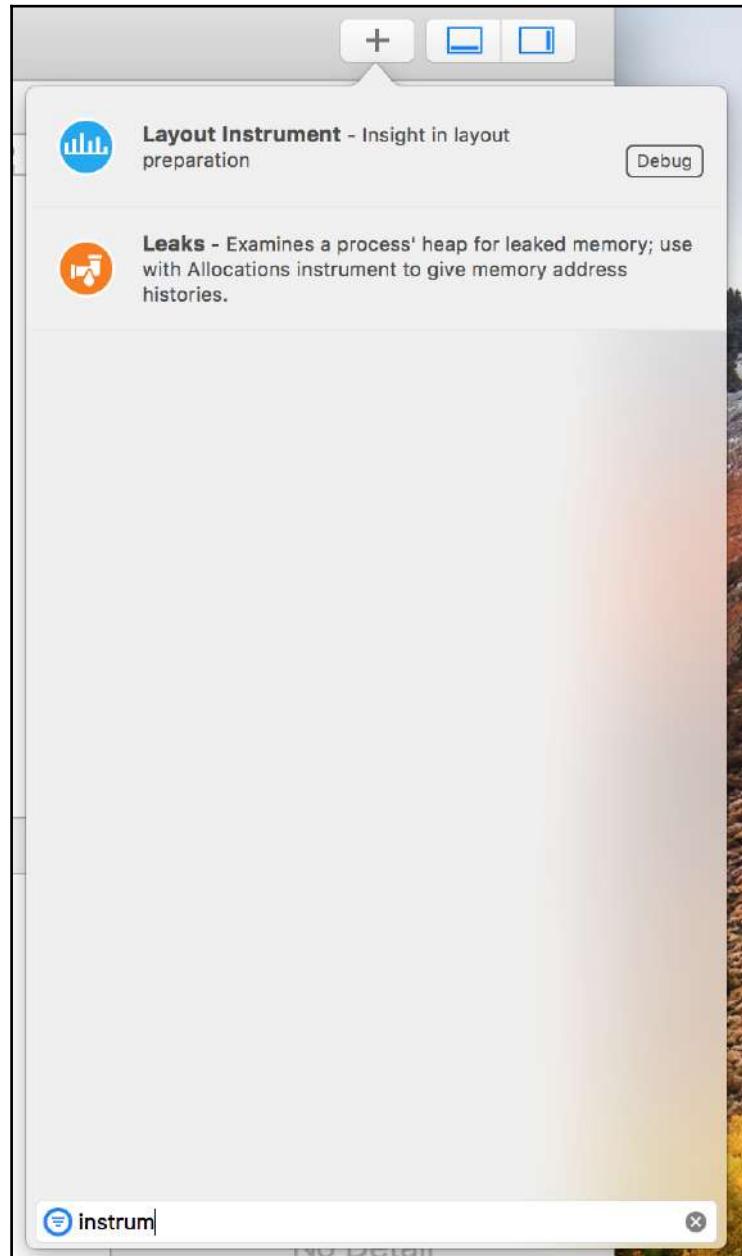
The preceding snippet defines all XML needed for a basic instrument. Again, there are four sections:

1. Metadata about the instrument.
2. Every instrument uses tables that refer to schemas. In this case, a table is created that uses the signpost-schema schema for its data.
3. This section defines how the custom instrument should be drawn on the timeline. Every graph uses a table for its underlying data and then extracts the required data from the table. Note that you didn't manually define duration on the signpost schema. You get this property for free from the schema.
4. A definition is given for how the instrument should be displayed in the detail area. The format of the detail area is a table, so in this section, you define the detail area in a table-like structure.

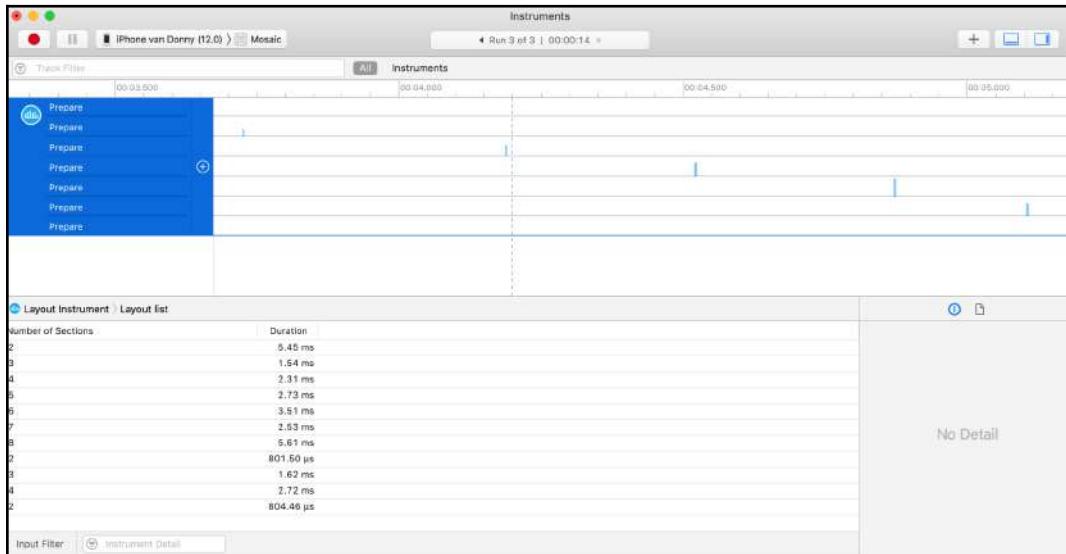
This is all the code you need to write a basic custom instrument. To test your instrument, select your Instrument Package target and use your Mac as the target define:



A special version of Instruments will launch. Pick a blank template and look for your custom instruments in the add menu:



Once you have found your instrument, add it to the timeline area and profile your app. You should see all your signpost data appear as expected. The detail area should now show a single entrance for every time you call `prepare()` so you can easily compare performance between similar calls:



This sample was quite simple but it does an excellent job of showing that you can do powerful things with custom instruments. If you want to learn even more about what kinds of custom instruments you can build and how you can do some custom processing on your signpost data, make sure to catch the *Creating Custom Instruments* talk from WWDC 2018.

Summary

You learned a lot about measuring your app's performance in this chapter. You also learned how to find common issues and how to use Instruments to figure out what's going on behind the scenes for your app. In your day-to-day development cycle, you won't use Instruments or Xcode's memory debugger very often. However, familiarizing yourself with these tools can save you hours of debugging. It can even help you to discover memory leaks or slow code before you ship your app.

Try to audit and measure several aspects of your app while you're developing it, and you can see how the performance of certain aspects of your app improves or degrades over time. This will help you to avoid shipping an app that's full of slow code or memory leaks. However, don't go overboard with optimizing until you encounter actual problems in your app. Prematurely optimizing your code often leads to code that is hard to maintain. The fixes you applied in this chapter are pretty simple; there were a few small bugs that could be fixed in just a few lines of code.

In addition to learning about existing instruments, you learned how to implement signposts in your app to measure your app's performance. You even learned how to create a custom instrument to visualize your signpost calls with.

Unfortunately, fixing issues in your app isn't always this easy. Sometimes, your code is as fast as it can be, yet it still takes too long and it makes scrolling your tables or collections choppy.

The next chapter will show you how to handle such cases by implementing asynchronous programming using operations and GCD.

Questions

1. What does the Time Profiler do?
 - a) It shows you when memory leaks occur.
 - b) It shows you memory usage over time.
 - c) It shows you how much time is spent in certain methods.
2. What does the Allocations instrument do?
 - a) It shows you your app's memory usage over time.
 - b) It shows you which objects reference another object.
 - c) It shows you how much time is spent in certain methods.
3. Which of the following does not necessarily indicate a memory leak?
 - a) Growing memory in the memory graph.
 - b) Seeing instances of view controllers that were popped off the navigation stack.
 - c) When no memory is freed up after your app receives a memory warning.

4. What does Xcode's memory debugger show you?
 - a) Corrupt addresses in memory.
 - b) The relationship between different objects that are in memory.
 - c) Deallocated objects.
5. What is signpost logging?
 - a) A low-overhead log to measure your app's performance.
 - b) Logs that show the flow of your app.
 - c) Logs to discover memory leaks.
6. Which elements does an Instruments Package have?
 - a) Metadata, a schema, and the instrument itself.
 - b) A schema, import statements for existing instruments, and a graph.
 - c) Signposts, tables, and a graph.

Further reading

- Creating Custom Instruments (WWDC 2018): <https://developer.apple.com/videos/play/wwdc2018/410/>
- Measuring Performance Using Logging (WWDC 2018): <https://developer.apple.com/videos/play/wwdc2018/405/>
- Instruments Developer Help: <https://help.apple.com/instruments/developer/mac/10.0/>

28

Offloading Tasks with Operations and GCD

In the previous chapter, you learned how to use Instruments to measure your app's performance. Measuring performance is a vital skill in discovering slow code or memory leaks. You saw that sometimes it's easy to fix slow code and increase performance by fixing a small programming error. However, the fix isn't always that easy.

An example of code that is very hard to make fast is networking code. When you fetch data from the network, you do this asynchronously. If networking code was not asynchronous, the execution of your app would halt until the network request is finished. This means that your app would freeze for a couple of seconds if the network is slow.

Another example of relatively slow code is a data file from the app bundle and decoding the raw data into an image. For small images, this task shouldn't take as long as a networking request, but imagine loading a couple of larger images. Decoding a large amount of data would take long enough to significantly slow down an app's scrolling performance, or render the interface unresponsive for longer than you should be comfortable with.

Making code run asynchronously is not very hard if you do it through dispatch queues. In Objective-C and early versions of Swift, this was mostly known as **GCD** or **Grand Central Dispatch**, and the methods you'd use to interact with GCD were ugly. Nowadays, you use the `DispatchQueue` class, and the methods you interact with are much cleaner.

This chapter will teach you how to make use of dispatch queues to write asynchronous code that can perform slow operations off the main thread, meaning that they don't block the interface while they're running. Once you've got the hang of using dispatch queues, you will abstract some of your asynchronous code into Operations, so it's possible to reuse them.

This chapter covers the following topics:

- Writing asynchronous code with dispatch queues
- Creating reusable tasks with operations

By the end of this chapter, you'll be able to enhance your apps by optimizing your code for asynchronous programming. You'll also be able to abstract specific tasks into Operations to make your app more straightforward to understand and maintain.

Writing asynchronous code

In Chapter 15, *Syncing Data with CloudKit*, you had an encounter with asynchronous code and multithreading. That chapter didn't go into much detail regarding multithreading and asynchronous code because the subject of threading is rather complex, and it's much more suited to a thorough discussion, which we're exploring in this chapter.

If you're unclear on what has already been explained, feel free to go back to Chapter 15, *Syncing Data with CloudKit*, to review the information presented there. The biggest takeaway from that chapter is that data fetching is performed on a background thread to avoid blocking the main thread. Once a fetch request is done, a callback function is executed, which allows you to use the main thread to update the user interface.

Understanding threads

You've seen the term thread a couple of times now, but you never explored them; you never really learned what a thread is or how a thread works. This section aims to make the subject of threading a lot clearer to you so you can fully understand what a thread is, and why threads are such a vital part of building apps.

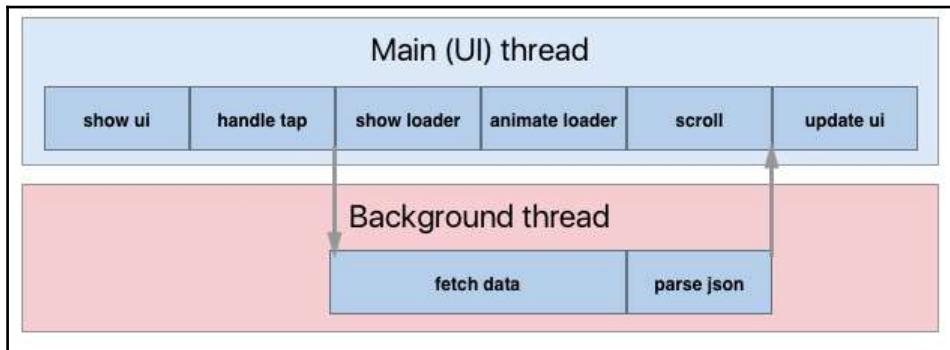
A good way to think of a thread is as a stack of instructions. In iOS, your app typically starts off with a single thread – the main thread. This thread is also known as the UI thread. It's called the UI thread because the main thread is where all of the user interface elements are configured, rendered, and pushed to the screen. Anything that is related to the user interface must be executed on the main thread, so if you think of a thread as a stack of instructions that are run one by one, it's easy to see why it's so important that the main thread doesn't get stuck performing a very slow instruction:



The preceding image shows a timeline where all code is executed on the main thread. Notice how the interface can't be updated until the **fetch data** and **parse JSON** instructions are completed. Also, note that fetching data takes a lot longer than displaying the interface or handling a tap. During the fetch data instruction, the app is unable to update any user interface elements or process any gestures or taps. This means that the app is practically frozen until the data is fetched and parsed.

A good, responsive application can't afford to wait for slow instructions. The interface should always respond to user input; if this isn't the case, the app feels slow, buggy, choppy, and just all-around bad.

If an app uses multiple threads, it can run various instruction stacks at the same time. Each *stack* is called a *thread*, and specific instructions can be performed on a different thread to ensure that the main thread remains responsive:



This second figure shows a more desirable scenario. The main thread only handles tasks that are related to the user interface, such as taps, animations, and scrolling. The background thread takes care of the tasks that are not related to the user interface and could potentially take a while to finish. By removing these instructions from the main thread and placing them on a different thread, like iOS does by default for networking, you can ensure that your app remains responsive, even if the network requests take several seconds to finish or never finish at all. Your app can utilize a large number of threads for different tasks.



The number of threads isn't infinite, so make sure that you optimize your code as much as possible to avoid locking up several threads with slow code.

In Chapter 27, *Discovering Bottlenecks with Instruments*, you used Instruments to locate slow code in a sample app. The slow code was an instruction on the main thread that took a very long time to complete, resulting in a frozen interface. However, threading would not have solved this particular issue. The slow code you discovered calculated the layout for a collection view. A collection view can't be rendered without calculating the layout first, so this is a scenario where it's essential to make sure that you write optimized code, instead of relying on threads for anything that's slow.

Now that you have a better understanding of threads and how they can be used, let's have a look at how to offload tasks to different threads.

Using dispatch queues in your application

A basic understanding of threads is good enough for you to start using them in your applications. However, once you start using them, there's a good chance that they suddenly become confusing again. If this happens to you, don't worry; threading is not easy. Now let's look at an example of threaded code:

```
var someBoolean = false

DispatchQueue(label: "MutateSomeBoolean").async {
    // perform some work here
    for i in 0..<100 {
        continue
    }

    someBoolean = true
}
```

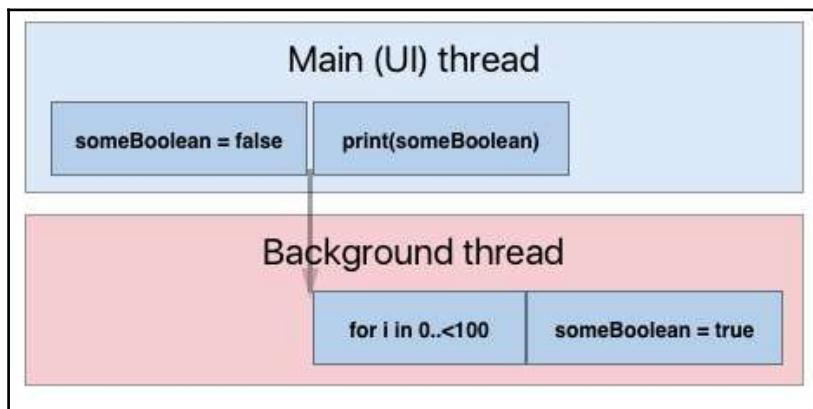
```
    }  
  
    print(someBoolean)
```

The preceding snippet demonstrates how you could mutate a variable after performing a task that is too slow to execute on the main thread. In the preceding code, an instance of `DispatchQueue` is created, and it's given a label. This creates a new queue on which you can execute instructions. The queue represents the background thread from the visualization you looked at earlier.

Then, the `async` method on `DispatchQueue` is called with the closure that should be executed on the freshly created queue. The loop inside of this block is performed on the background thread. In the visualization from before, this would roughly compare to the fetch data and parse JSON instructions. Once the task is done, `someBoolean` is mutated.

The last line in the snippet prints the value of `someBoolean`. What do you think the value of `someBoolean` is at that point? If your answer is `false`, good job! If you thought `true`, you're not alone. A lot of people who start writing multithreaded, asynchronous code don't immediately grasp how it works.

The following image shows what the code does in the background. This should make what happens and why `someBoolean` was `false` when it got printed more obvious:



Because this code uses a background thread, the main thread can immediately move to the next instruction. This means that the for loop and the print run simultaneously. In other words, `someBoolean` is printed before it's mutated on the background thread. This is both the beauty and a caveat of using threads. When everything starts running simultaneously, it is hard to keep track of when something completes.

The preceding visualization also exposes a potential problem in the code. A variable was created on the main thread, and then it got captured in the background thread which then mutated the variable. Doing this is not recommended; your code could suffer from unintended side-effects, such as race conditions, where both the main thread and the background thread mutate a value, or worse, you could accidentally try to access a Core Data object on a different thread than the one it was created on. Core Data objects do not support being used on multiple threads so you should always try to make sure that you avoid mutating or accessing objects that are not on the same thread as the one where you access them.

So, how can you mutate `someBoolean` safely and print its value after mutating it? Well, you could use a callback closure to achieve this. The following code is a sample of what that would look like:

```
func executeSlowOperation(withCallback callback: @escaping ((Bool) -> Void)) {
    DispatchQueue(label: "MutateSomeBoolean").async {
        // perform some work here
        for i in 0..<100 {
            continue
        }

        callback(true)
    }
}

executeSlowOperation { result in
    DispatchQueue.main.async {
        someBoolean = result
        print(someBoolean)
    }
}
```

In this snippet, the slow operation is wrapped in a function that is called with a callback closure. Once the task is complete, the callback is executed, and it is passed the resulting value from the background thread. The closure makes sure that its code is executed on the main thread. If you don't do this, the closure itself would have been executed on the background thread. It's important to keep this in mind when calling asynchronous code.

The callback-based approach is excellent if your callback should be executed when a single task is finished. However, there are scenarios where you want to complete some tasks before moving to the next task. You have already used this approach in [Chapter 14, Being Proactive with Background Fetch](#).

Let's review the heart of the background fetch logic that was used in that chapter:

```
func application(_ application: UIApplication,
performFetchWithCompletionHandler completionHandler: @escaping
(UIBackgroundFetchResult) -> Void) {

    let fetchRequest: NSFetchedResultsController<Movie> = Movie.fetchRequest()
    let managedObjectContext = persistentContainer.viewContext
    guard let allMovies = try? managedObjectContext.fetch(fetchRequest)
    else {
        completionHandler(.failed)
        return
    }

    let queue = DispatchQueue(label: "movieDBQueue")
    let group = DispatchGroup()
    let helper = MovieDBHelper()
    var dataChanged = false

    for movie in allMovies {
        queue.async(group: group) {
            group.enter()
            helper.fetchRating(forMovieId: movie.remoteId) { id,
                popularity in
                guard let popularity = popularity, popularity !=
                    movie.popularity else {
                    group.leave()
                    return
                }

                dataChanged = true

                managedObjectContext.persist {
                    movie.popularity = popularity
                    group.leave()
                }
            }
        }
    }

    group.notify(queue: DispatchQueue.main) {
        if dataChanged {
            completionHandler(.newData)
        } else {
            completionHandler(.noData)
        }
    }
}
```

When you first saw this code, you were probably able to follow along, but it's unlikely that you were completely aware of how complex this method is. Multiple dispatch queues are used in this snippet. To give you an idea, this code starts off on the main thread. Then, for each movie, a background queue is used to fetch its rating. Once fetching all these ratings is complete, the managed object context's dispatch queue is used to update the movie. Think about all this switching between dispatch queues that is going on for a second. Quite complex, isn't it?

The background fetch method needs to call a completion handler when it is done fetching all the data. However, a lot of different queues are used, and it's kind of hard to tell when all the fetch operations have completed. This is where dispatch groups come in. A dispatch group can hold onto a set of tasks that are executed either serially or in parallel.

When you call `enter()` on a dispatch group, you are also expected to call `leave()` on the group. The `enter` call tells the group that there is unfinished work in the dispatch group. When you call `leave()`, the task is marked as completed. Once all tasks are completed, the group executes a closure on any thread you desire. In the example, `notify(queue:)` is the method used to perform the completion handler on the main queue.

It's okay if this is a bit daunting or confusing right now. As mentioned before, asynchronous programming and threads are pretty complex topics, and dispatch groups are no different.



The most important takeaways regarding dispatch groups are that you call `enter()` on a group to submit an unfinished task, you call `leave()` to mark the task finished, and you use `notify(queue:)` to execute a closure on the queue passed to this method once all tasks are marked as completed.

The approach you've seen so far makes direct use of closures to perform tasks. This causes your methods to become long and relatively complex since everything is written in line with the rest of your code. You already saw how mixing code that exists on different threads can lead to confusion because it's not very obvious which code belongs on which queue. Also, all this inline code is not particularly reusable. You can't pick up a particular task and execute it on a different queue for instance, because the code is tightly coupled to a specific dispatch queue.

You can use `Operations` to make tasks that are easy to reuse and decoupled from running on a specific queue.

Creating reusable tasks with Operations

You have just learned about `DispatchQueues`, and how they are used to schedule tasks that need to be performed on a different thread. You saw how this speeds up code and how it avoids blocking the main thread. In this section, you will take this one step further. The first reason for this is because asynchronous work is better organized if it is implemented as an isolated object that can be scheduled for execution instead of having several closures that make your code less readable.

The solution to having closures all over your asynchronous code is to use `Operation` instead of a closure. And instead of queueing everything in a dispatch queue, `Operation` instances are queued on `OperationQueue`. `OperationQueue` and `DispatchQueue` are similar, but not quite the same. `OperationQueue` can schedule `Operations` on one or more `DispatchQueues`. This is important because of the way in which `Operations` work.

When using `OperationQueue`, you can execute `Operations` in parallel or serially. It is also possible to specify dependencies for `Operations`. This means that you can make sure that certain `Operations` are completed before the next operation is executed. `OperationQueue` will manage the `DispatchQueues` needed to make everything happen, and it will execute `Operations` in the order in which they become ready to execute. Moreover, code that uses `Operations` will typically be easier to maintain and understand because all work is beautifully encapsulated in isolated building blocks.

The next section will briefly cover some of the basic concepts of `Operations`.



If you're looking to learn more about using `Operations` in interesting and advanced ways, make sure to check out *Apple's Advanced NSOperations talk* from WWDC 2015. All code for this talk is presented in Swift 2.0, so you'll need to translate this code to Swift 4.2 yourself, but it's definitely worth a watch.

Using Operations in your apps

Let's take a deep dive into Operations and refactor the background fetch code from the **MustC** app to make it use Operations. To do this, you will create two Operation subclasses: one that fetches data and updates a `movie` object, and one that calls the completion handler once all other operations are complete.

The setup will have a single `OperationQueue` that will execute all of the instances of the `fetch` Operation subclass and a single operation that calls the background fetch completion handler. The completion operation will have all of the fetch operations as its dependencies, so it's automatically executed when all fetch operations are completed.

Whenever you create an `OperationQueue` instance, you can specify the number of concurrent Operation instances that can be executed on the queue. If you set this number to one, you have a serial queue that runs all operations one by one in the order in which they become ready to execute.

An operation is considered ready when all preconditions for the operation are met. A great example of this is dependencies. An operation with dependencies is not ready to execute until all of the operations that it depends on are completed. Another example is exclusivity. It's possible to configure your setup to allow only a single operation of a particular type to be running at a time. An operation that is set up like that won't become ready until no other operations of that type are running.

If you set the maximum number of concurrent operations to a higher number, it's not guaranteed that the queue will run that number of operations simultaneously. Imagine setting the maximum amount to 1,000, and you put 2,000 operations in the queue. It's not likely that you will see 1,000 operations running in parallel. The system ultimately decides how many operations it will run at the same time, but it's never more than your maximum value.

As mentioned before, you can improve the **MustC** app by using `OperationQueue`. Get started with this refactor by creating an `OperationQueue` instance. You will add all download operations to this queue. Replace the implementation of `application(_:performBackgroundFetchWithCompletionHandler:)` in `AppDelegate` with the following:

```
func application(_ application: UIApplication,  
performFetchWithCompletionHandler completionHandler: @escaping  
(UIBackgroundFetchResult) -> Void) {  
  
    let queue = OperationQueue()
```

```
let fetchRequest: NSFetchedRequest<Movie> = Movie.fetchRequest()
let managedObjectContext = persistentContainer.viewContext
guard let allMovies = try? managedObjectContext.fetch(fetchRequest)
else {
    completionHandler(.failed)
    return
}
}
```

The preceding code creates an operation queue. This is the queue that will be used to fetch all movies and call the background fetch completion handler once all downloads are done. Before you can add operations to the queue, you must create the appropriate classes to contain your operation. Create a new file and name it `UpdateMovieOperation.swift`.

Every custom operation you create should subclass the `Operation` base class. The `Operation` class implements most of the glue and boilerplate code involved in managing and executing operations and queues. There are a couple of mandatory read-only variables that you need to implement for your operation. Also, you need to make sure these variables are **KVO (Key-Value Observing)** compliant, so the operation queue can observe the various states your operation will go through during its lifetime. Add the following basic implementation for `UpdateMovieOperation`:

```
import Foundation

class UpdateMovieOperation: Operation {
    override var isAsynchronous: Bool { return true }
    override var isExecuting: Bool { return _isExecuting }
    override var isFinished: Bool { return _isFinished }

    private var _isExecuting = false
    private var _isFinished = false
    var didLoadNewData = false

    let movie: Movie

    init(movie: Movie) {
        self.movie = movie
    }
}
```

You'll immediately notice that this implementation overrides a couple of variables. These are the read-only variables that were mentioned earlier. The `isExecuting` and `isFinished` variables return the value of two private variables that will be mutated to reflect the operation's state later.

Furthermore, the operation keeps track of whether new data was loaded, and there are a property and initializer to attach a `Movie` to the operation. So far, this operation isn't very exciting. Add the following methods to your operation so it can do something:

```
override func start() {
    super.start()

    willChangeValue(forKey: #keyPath(isExecuting))
    _isExecuting = true
    didChangeValue(forKey: #keyPath(isExecuting))

    let helper = MovieDBHelper()
    helper.fetchRating(forMovieId: movie.remoteId) { [weak self] id,
        popularity in
        defer {
            self?.finish()
        }

        guard let popularity = popularity,
              let movie = self?.movie,
              popularity != movie.popularity
        else { return }

        self?.didLoadNewData = true

        movie.managedObjectContext?.persist {
            movie.popularity = popularity
        }
    }
}

func finish() {
    willChangeValue(forKey: #keyPath(isFinished))
    _isFinished = true
    didChangeValue(forKey: #keyPath(isFinished))
}
```

Apple's guidelines state that every subclass of `Operation` should override the `start()` method and initiate the operation from there. You'll note that the superclass implementation is called first. This is because the superclass takes care of several under-the-hood tasks that must be performed to make the operation work. Next, `willChangeValue(forKey:)` and `didChangeValue(forKey:)` are called to fire the KVO notifications, so the operation queue knows when the state of the operation has changed or is about to change.



You might notice that the values of `isFinished` and `isExecuting` are not changed after calling `willChangeValue(forKey:)`. This is okay because these methods only tell any observers that reading specific properties after `didChangeValue(forKey:)` will yield a different value than before. Since `isFinished` and `isExecuting` return the value of the private properties that are changed, `isFinished` and `isExecuting` will return different values, as the observer expects.

Next, the code from before is used to fetch and update the movie. A `defer` block is used to call the `finish()` method when the operation is done, regardless of how the network request went. By using `defer` instead of manually calling `finish()` when appropriate, you can't forget to call `finish()` if the code changes at some point. The `finish()` method makes sure that the operation queue is notified about the operation's new status by firing the corresponding KVO notifications.

Another operation that calls the background fetch completion handler should be created. This operation should loop through all of its dependencies, check whether it's a movie update operation, and if it is, it should check whether at least one of these operations has loaded new data. After doing this, the completion handler should be called with the corresponding result, and finally, the operation should finish itself. Create a new file in the `Operations` folder and name it

`BackgroundFetchCompletionOperation`. Add the following implementation:

```
import Foundation
import UIKit

class BackgroundFetchCompletionOperation: Operation {
    override var isAsynchronous: Bool { return true }
    override var isExecuting: Bool { return _isExecuting }
    override var isFinished: Bool { return _isFinished }
    var _isExecuting = false
    var _isFinished = false

    let completionHandler: (UIBackgroundFetchResult) -> Void

    init(completionHandler: @escaping (UIBackgroundFetchResult) -> Void)
    {
        self.completionHandler = completionHandler
    }

    override func start() {
        super.start()
    }
}
```

```
willChangeValue(forKey: #keyPath(isExecuting))
_isExecuting = true
didChangeValue(forKey: #keyPath(isExecuting))

var didLoadNewData = false

for operation in dependencies {
    guard let updateOperation = operation as? UpdateMovieOperation
else { continue }

    if updateOperation.didLoadNewData {
        didLoadNewData = true
        break
    }
}

if didLoadNewData {
    completionHandler(.newData)
} else {
    completionHandler(.noData)
}

willChangeValue(forKey: #keyPath(isFinished))
_isFinished = true
didChangeValue(forKey: #keyPath(isFinished)) }
```

The implementation for this operation is pretty similar to the movie update operation. The operation should be initialized with the completion handler that was passed to the background fetch method in `AppDelegate`, and it's called after figuring out whether new data was fetched by looping through all movie update operations. Let's see how all this comes together by updating the background fetch logic in `AppDelegate`. Add the following code to the

`application(_:performFetchWithCompletionHandler:)` method, right after fetching the movies:

```
let completionOperation =
BackgroundFetchCompletionOperation(completionHandler:
completionHandler)

for movie in allMovies {
    let updateOperation = UpdateMovieOperation(movie: movie)
    completionOperation.addDependency(updateOperation)

    queue.addOperation(updateOperation)
```

```
    }  
  
    queue.addOperation(completionOperation)
```

This code is a lot more readable than what was there before. First, the completion operation is created. Next, an update operation is created for each movie, this operation is added as a dependency for the completion operation, and it's added to the operation queue. Once all dependencies are set up, the completion operation itself is added to the queue as well. All of the movie update operations will automatically start executing simultaneously, and once they're all done, the completion operation becomes ready to execute. Once this happens, the completion operation will run and the background fetch completion handler will be called.

Even though `Operations` involve a bit more boilerplate code regarding managing execution state, you do end up with code that is a lot easier to read and understand. This chapter only covered setting up dependencies for operation, but if you study *Apple's Advanced NSOperations* video that was mentioned earlier, you'll find that you can do powerful, complex, and amazing things with operations.

Summary

This chapter showed you that asynchronous code can be hard to understand and reason about, especially when a lot of code is running at the same time; it can be easy to lose track of what you're doing. You also learned that `Operations` are a convenient way to reduce complexity in your application, resulting in code that is easier to read, change, and maintain. When an operation depends on multiple other `Operations` to be completed, it can be extremely convenient to use `OperationQueue`, as it dramatically reduces the complexity of the code you write.

It's been mentioned before, but if you intend to make use of operations in your app, make sure to check out Apple's Demonstration of Advanced Operations from WWDC 2015. Operations are capable of far more than you've seen in this chapter, and it's highly recommended to see how Apple uses operations to create rock-solid apps. Once your app is covered by tests, measured with instruments, and improved with asynchronous code and operations, it's probably time that you share your app with others.

The next and final chapter of this book will show you how to set yourself up for deploying your app through TestFlight and the App Store.

Questions

1. What are the downsides of running certain code on the main thread?
 - a) It can block the UI.
 - b) It's against best practices.
 - c) The main thread can only handle UI code.
2. In which circumstances will threading not help your performance?
 - a) When you have a lengthy layout-related calculation.
 - b) When you load data from the network.
 - c) When you load many images from disk.
3. Which GCD feature can you use to execute an action when several other actions are done executing?
 - a) DispatchClub.
 - b) DispatchGroup.
 - c) GCD_Group.
4. Which of the following is not a benefit of operations?
 - a) More readable code.
 - b) Faster code execution.
 - c) Operations can depend on other operations.
5. What's the maximum number of Operations you can run at the same time?
 - a) There is no hard limit, iOS decides on the fly.
 - b) 4.
 - c) 10.
6. What is KVO?
 - a) A way for objects to observe properties of another object.
 - b) A way to boost the execution speed of threads.
 - c) A technique that is used to make operations depend on each other.
7. How do you start executing operations on an operation queue?
 - a) By calling `start()` on the queue.
 - b) By calling `resume()` on the queue.
 - c) By adding operations to it.

Further reading

- Modernizing Grand Central Dispatch Usage: <https://developer.apple.com/videos/play/wwdc2017/706/>
- Advanced NSOperations: <https://developer.apple.com/videos/play/wwdc2015/226/>

29

Submitting Your App to the App Store

Possibly the most exciting part of the development cycle is getting your app into the hands of some real users. The first step in doing so is usually to send out a beta version of your app so you can get feedback and gather some data about how your app is performing before you submit to the App Store and release your app to the world. Once you're satisfied with the results of your beta test, you must submit your app to Apple so they can review it before your app is released to the App Store.

In this chapter, you'll learn everything about packing up your app and submitting it to Apple's App Store Connect portal. Using App Store Connect, you can start beta-testing your app, and you can also submit it to Apple for review. App Store Connect is also used to manage your app's App Store description, keywords, promotional images, and more. This chapter will show you how to fill out all the information for your app correctly. You will go through the following steps in this chapter:

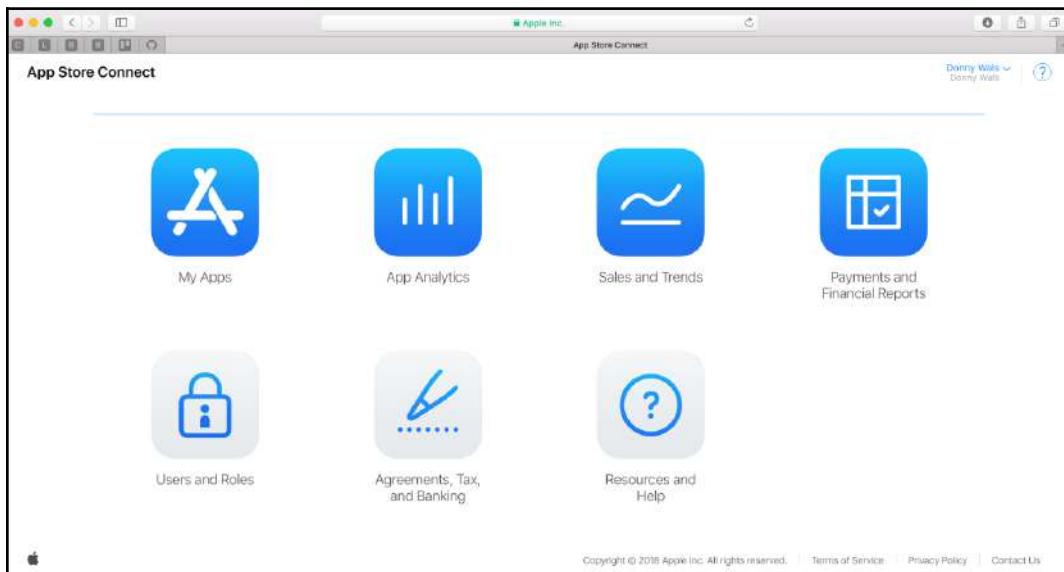
- Adding your application to App Store Connect
- Packaging and uploading your app for beta testing
- Preparing your app for launch

These steps closely resemble the process you'll go through when you're ready to launch your app. Let's get right to it, shall we?

Adding your application to App Store Connect

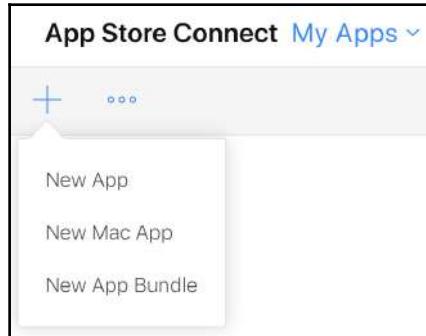
The first thing you're going to want to do when you're ready to release your app is to register your app with App Store Connect. To access App Store Connect, you must be enrolled in the Apple Developer program. You can do this through Apple's developer portal at <https://developer.apple.com>. After purchasing your membership, you can log in to your App Store Connect account on <https://appstoreconnect.apple.com> using your Apple ID.

After logging into your App Store Connect account, you are presented with a screen that has a couple of icons on it:



This screen is your portal to manage your App Store presence. From here, you can manage test users, track your app's downloads, monitor app usage, and more. But most importantly, it's where you create, upload, and publish your apps to Apple's beta-distribution program, called **TestFlight**, and once you're done beta testing, to the App Store. Go ahead and peek around a bit; there won't be much to see yet, but it's good to familiarize yourself with the App Store Connect portal.

The first step in getting your app out to your users is to navigate to the **My Apps** section. This section is where you'll find all the apps you have created and where you can add new apps. To add a new app, click the + icon in the top left and click **New App**:



After clicking this, you're presented with a window in which you can fill out all the necessary information about your app. This is where you pick your app's name, select the platform on which it will be released, and a few more properties:

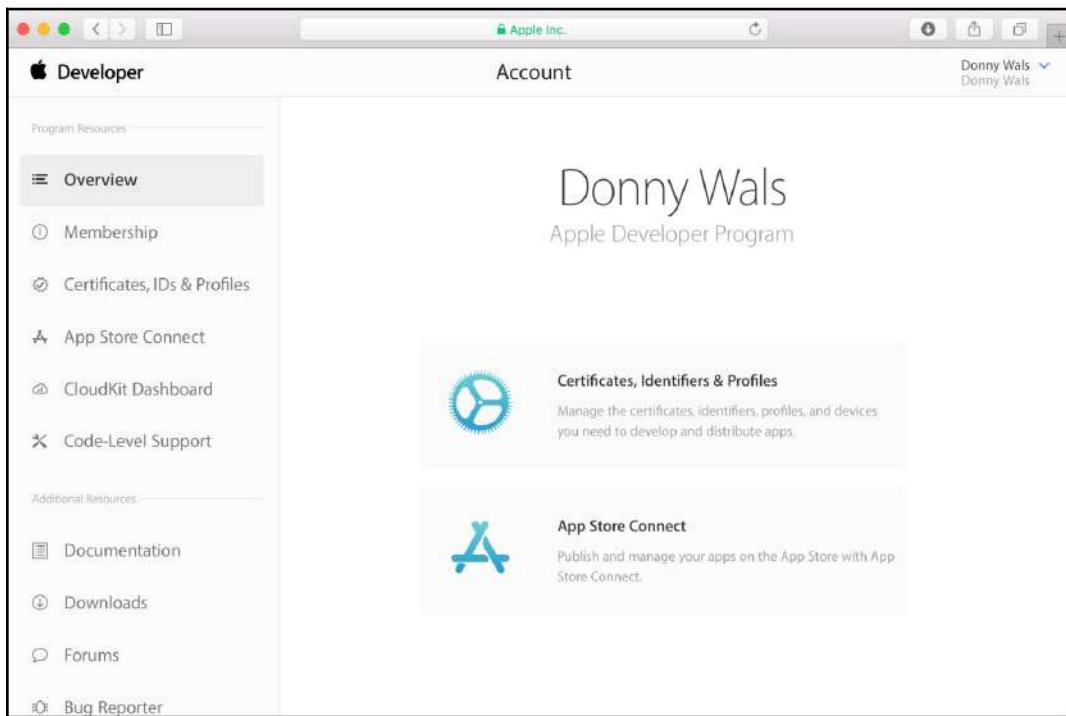
A screenshot of the "New App" configuration window. It has several fields:

- Platforms**: A section with checkboxes for "iOS" (which is checked) and "tvOS".
- Name**: A text input field containing "The DailyQuote".
- Primary Language**: A dropdown menu set to "English (U.S.)".
- Bundle ID**: A dropdown menu showing a partial value: "XC com.donnywals.MustC - com.donnywals.MustC".
- SKU**: A text input field containing "com.donnywals.MustC".

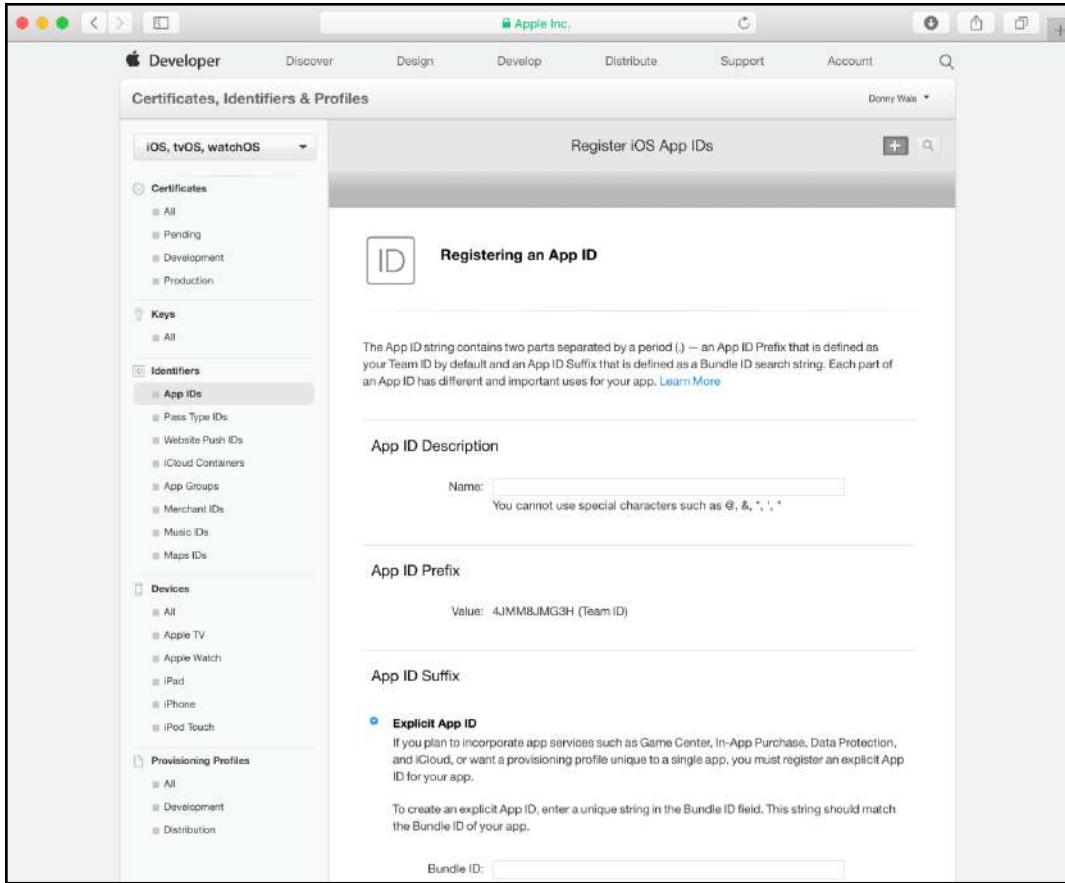
At the bottom right are two buttons: "Cancel" and "Create".

The Bundle ID field is a drop-down menu that contains all of the app IDs that you have registered for your team in Apple's developer portal. If you've been developing with the free-tier developer account up until the last minute, or you haven't used any special features such as push notifications or App Groups, it's possible that your app's Bundle ID is not in the drop-down menu.

If this is the case, you can manually register your Bundle ID in the developer portal. Navigate to <https://developer.apple.com/> in your browser and click the **Account** menu item. You can use the Account page to manage certificates, Bundle IDs, devices, and more. A lot of this is automatically taken care of by Xcode, but you'll occasionally find yourself in this portal. For instance to manually register your app's Bundle ID:



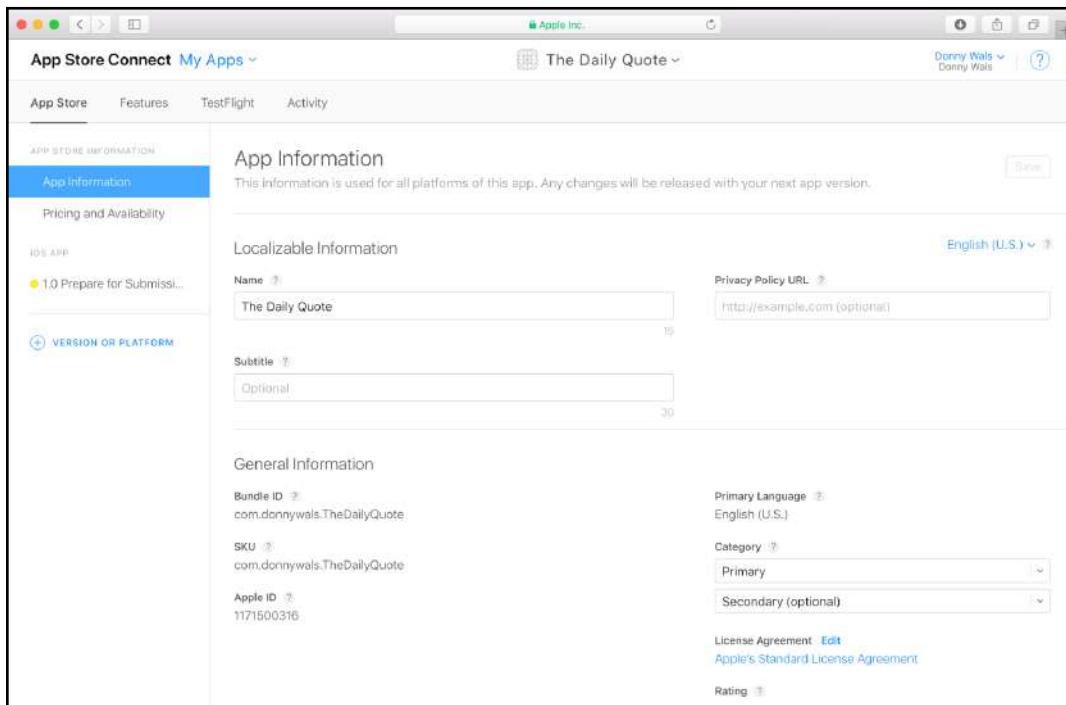
To register your Bundle ID, click on the **Certificates, IDs & Profiles** item on the left-hand side of the page. On the Certificates, IDs & Profiles page, click **App IDs** in the menu on the left-hand side. This will present you with a list of currently registered apps. In the top-right corner, there's a + icon. Click this to add a new ID to your profile:



To add your Bundle ID, all you need to do is fill in the form fields. You'll want to use a descriptive name for your app name. It can be the same as the name you have set for your app in Xcode, but it can also be a different name; it doesn't have to match. Make sure to select the **Explicit App ID** field and copy the Bundle ID from your Xcode project. It's important that you perfectly match this. If you don't do this, you'll run into issues later because your app can't be identified if the Bundle ID is incorrect.

Once you've done this, you can scroll all the way down and save your new ID. You don't have to select any capabilities since Xcode will automatically manage this for you when you enable or disable them in the **Capabilities** tab.

After manually registering your Bundle ID, you should be able to move back to App Store Connect, add a new app, and select your app's Bundle ID. After you've done this and you've created your app in the App Store Connect portal, have a look around in your app's settings. There are a lot of form fields that you can fill in. The first screen you'll see is the **App Information** screen. This is where you fill out the information about your app, assign it a localized name that appears in the App Store, and assign categories to your app:



Next, there's the **Pricing and Availability** screen. This is where you decide in which countries your app can be downloaded and also how much it costs. Lastly, there is the **Prepare for Submission** menu item.

Whenever you add a new version of your app, you should fill out the form fields on this screen, and there are quite a lot of them. The **Prepare for Submission** form is used to provide screenshots, keywords, a description for your app, privacy policies, and more. Go ahead and have a look at what's in there. Luckily, everything you have to fill in is pretty straightforward.

Once you have registered your app on App Store Connect, you can upload your app. To do this, you use Xcode to package up your app and send it off to App Store Connect.

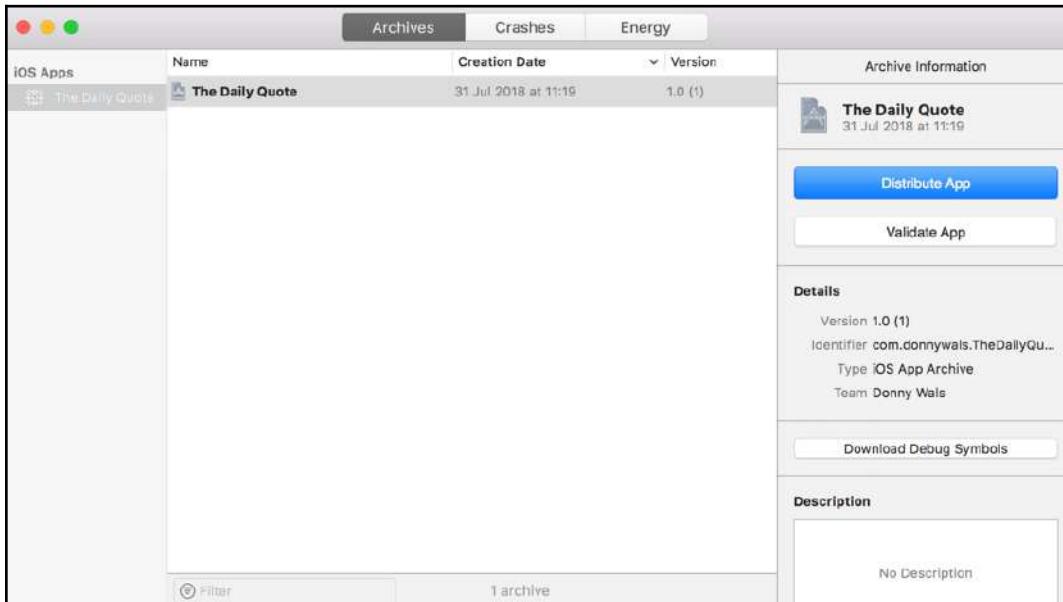
Packaging and uploading your app for beta testing

To send your app out to beta testers and eventually real users, you must first archive your app using Xcode. Archiving your app will package up all contents, code, and assets. To archive your app, you must select **Generic iOS Device** from the list of devices your app runs on in Xcode:



With this build device selected, select **Product | Archive** from the top menu in Xcode. When you do this, a build will start that's a lot slower than usual. That's because Xcode is building your app in release mode so it is optimized and can run on any device.

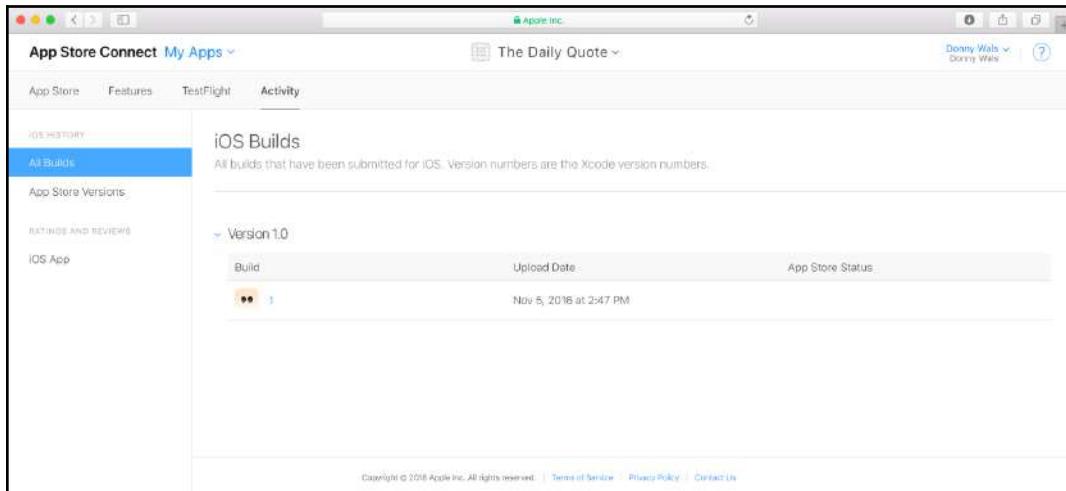
Once the archive is created, Xcode will automatically open the organizer panel for you. In this panel, you get an overview of all apps and archives that you have created:



Before you archive your app, you should make sure that your app is ready for release. This means that you must add all of the required app icon assets to the `Images.xcassets` resource. If your app icon set is incomplete, your app will be rejected when you try to upload it to App Store Connect, and you'll have to generate your archive all over again.

When you're ready to upload your build to App Store Connect to send it to your beta testers and eventually your users through the App Store, you should select your latest build and click the **Upload to App Store** button. A popup will appear to guide you through a couple of settings. You should use the default settings most of the time and click **Next**. Finally, your app will be verified based on metadata and uploaded to App Store Connect. If there are errors in your archive, such as missing assets or unsigned code, you'll learn about it through error messages in the upload popup window. When your upload succeeds, you'll also be notified through the popup.

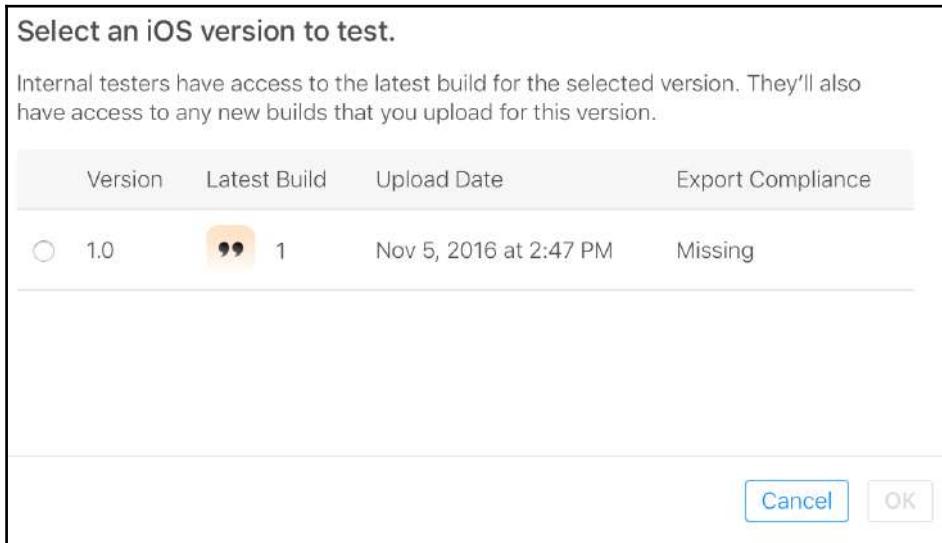
Once you've uploaded your build, you can go to the activity panel for your app in App Store Connect. This is where you'll see your build's current status. If you have just uploaded a build, its status will be **Processing**. Sometimes this step takes a while, but usually no longer than a couple of hours:



While the app is processing, you can start to prepare your **TestFlight** settings. Select the **TestFlight** menu item and fill in the Test Information form. If you're rolling out your beta test internationally, you might want to provide information in multiple languages, but you don't have to.

Next, select the **Internal Testing** menu item in the sidebar on the right. In this panel, you can select users that are added to your account for beta testing. This type of testing is often the first type of testing you do, and it's mostly intended for testing apps internally with members of your team or with close friends and family. You can add more internal users to your account through the **Users and Roles** section in App Store Connect.

Once your app is processed, you'll receive an email, and you can select which version of your app should be used for internal testing:



Once you've added testers and you've selected a build to test, you can click the **Start Testing** button to send out a beta invite to your selected testers. They will receive an email that enables them to download your app through the **TestFlight** app for iOS.

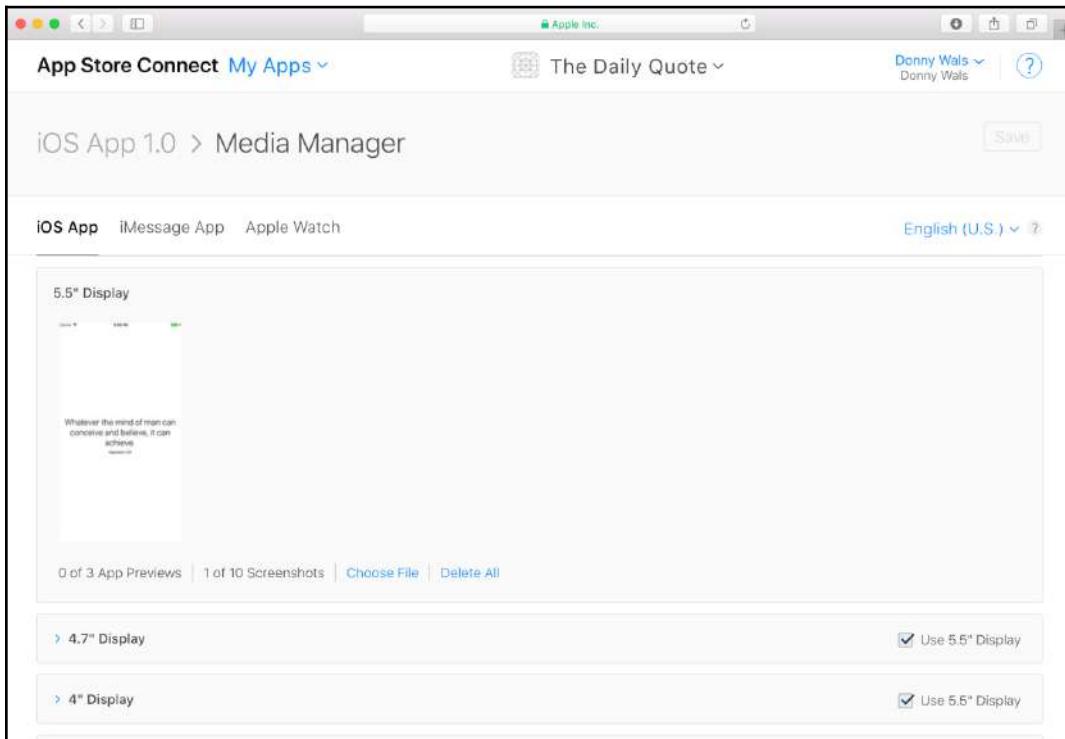
Once your internal testers are happy with your app, you can select some external beta testers for your app. **External testers** are typically people who aren't in your team or organization, such as existing users of your app or a selection of people from your app's target audience.

Setting up an external beta test is done identically to how you set up the internal test. You can even use the same build that you used for internal testing for external testing. However, external tests typically require a quick review by Apple before invites can be sent out. These reviews don't take long and passing the beta test review does not mean you'll also pass the App Store review.

This is all you need to know about setting up a beta test through **TestFlight**. When you're happy with the results of your beta test and your app has passed the real-world test, it's time for you to prepare to release your app into the wild through the App Store.

Preparing your app for launch

Moving from beta testing to releasing your app does not require much effort. You use the same version of your app as you've already exported and tested with your users. To be able to submit your app for review by Apple, you have to add more information about your app, and you should set up your App Store presence. The first thing you should do is create a couple of screenshots of your app. You add these screenshots to your App Store page, and they should look as good as possible because potential users will use screenshots to determine whether they want to buy or download your app. The simplest way to create screenshots is to take them on a 5.5-inch iPhone and a 12.9-inch iPad. You can provide screenshots for every type of device that exists, but you must at least provide them for a 5.5-inch iPhone and a 12.9-inch iPad. You can use the **Media Manager** feature in App Store Connect to upload the large-sized media and have it scale down for smaller devices:



After submitting screenshots, you should also fill out a description and keywords for your application. Make sure that your app description is clear, concise, and convincing. You should always try to use as many keywords as you can. Apple uses your keywords to determine whether your app matches search queries from users. Try to come up with synonyms or words you would look for when you'd search for an app that does what your app does.

If your app features an iMessage or Watch app, you should upload screenshots for these apps as well. You can't provide separate keywords or descriptions for these extensions, but they will have their own image galleries in the App Store.

The next step in the submission form is to select your app binary and provide some general information about the app and the person responsible for publishing the app. Usually, you'll want to choose the version of the app you've been beta testing up to the point of release.

Lastly, you must provide some information to Apple about how your app should be reviewed. If your app required a demo account, provide credentials to the reviewer. If your app has been rejected before due to something being unclear, it's usually a good idea to clarify the past misunderstanding in the notes section. This has proven to help for some apps, resulting in accepted reviews at the first try rather than being rejected and providing explanations afterward. When everything is filled out, hit the **Save** button to store all of the information you just entered. Then, if you're satisfied with everything and ready to take the final leap toward releasing your app, press **Submit for Review** to initiate the review process.

Getting your app reviewed by Apple can take from just a single day to a couple of days or even longer. Once you have submitted your app, it's important that you patiently wait until you hear from Apple. Sending them queries about reviewing your app faster or asking them about the current status often yields no results, so there's no point in trying to push Apple for a review.



If you do need to get your app reviewed and released fast and you have a legitimate reason, you can always apply for expedited review. If Apple agrees that a faster review will benefit not just you but also your users, your app could be reviewed in a matter of hours. Note that you should not abuse this. The more often you apply for an expedited review, the less likely Apple is to grant you an exception. Expedited reviews should only be requested in exceptional cases.

Well, time to get some cocoa, coffee, tea, or whatever you prefer to drink. You can now sit back for a while and wait while Apple reviews your app so you can release it to the App Store.

Summary

This final chapter covered preparing to release your app. You learned how to archive and export your app. You saw how to upload it to App Store Connect and how to distribute your app as a beta release. To wrap everything up, you saw how to submit your app for review by Apple to release it to the App Store. Releasing an app is exciting; you don't know how well your app will perform or whether people will enjoy using it. A good beta test will help a lot, you'll be able to spot bugs or usability issues, but there's nothing like having your app in the hands of actual users.

Most developers invest a lot of time and effort into building their apps, and you are one of them. You picked up this book and went from an iOS enthusiast to an iOS master who knows exactly how to build great apps that make use of iOS' newest and greatest features. When you're ready to launch your own app into the App Store, you'll learn how exciting and nerve-racking it can be to wait for Apple to review and hopefully approve your app. Maybe you get rejected on your first try; that's possible. Don't worry too much about it; even the biggest names get rejected sometimes, and often fixing the reason isn't too complicated. Do make sure to read the App Store review guidelines before you submit; these guidelines give a pretty good indication about what you can and can't do in your apps.

Since this is the last chapter in this book, I would like to sincerely thank you for picking up this book and using it as one of the stepping stones toward becoming a master of iOS programming. I hope to have left you with the skills required to venture out on your own, read Apple's documentation, and build amazing applications. Thanks again, and if you've created something cool using this book, please feel free to reach out to me. I would love to see your application.

Further reading

- **App Store Review Guidelines:** <https://developer.apple.com/app-store/review/guidelines/>

Appendix

Chapter 12

1. What does a persistent store coordinator do?
 - b) It talks to the persistent store.
2. What is the function of managed object context?
 - b) It mediates between managed objects and the persistent store coordinator.
3. How many managed object contexts can you use in an app?
 - c) An unlimited amount.
4. What is the correct way to call `save()` on a managed object context?
 - c) `try save()`.
5. When should you use a fetched results controller?
 - b) Only when you need to fetch data from the database and want to react to changes.
6. Does a fetched results controller always go to the database to fetch data?
 - c) Yes, all fetch requests must go to the database.

Chapter 13

1. What object should you use to make web requests?
 - a) URLSession.
2. How are web requests executed?
 - a) Asynchronously.
3. What is a typealias?
 - c) An alias for any other type, for instance a callback.
4. What happens if you use managed objects on the wrong thread?
 - c) Your app might crash.
5. Which object do you need if you make an HTTP POST request?
 - c) POSTRequest.
6. What is ATS?
 - b) Apple's way of enforcing secure web requests.

Chapter 14

1. For how long can you perform work when your app is woken up for background refresh?
 - b) There is no way to be sure.
2. How do you enable background fetch for your app?
 - c) Both.
3. Which statement about the minimum refresh interval is true?
 - b) iOS will determine when it's the best time wake up your app. It will be no more often than you specify.

4. What did you use a dispatch group for in this chapter?
 - b) To be notified when all network calls were done.
5. When should you refactor your code?
 - c) Any time you think you can improve something.
6. What do you call the ID that is passed to the movieById case on the MovieDBHelper.Endpoint enum?
 - a) Associated value.

Chapter 15

1. How does the CloudKit server notify apps about updates?
 - b) Silent push notifications.
2. Which object do you send along with a fetch request to CloudKit so you only receive new changes?
 - a) A change token.
3. How does your code know whether a certain CKRecord is a movie, family member, or something else?
 - c) Through recordType.
4. What kind of data do you have to store alongside your objects in Core Data to be able to properly sync local data with CloudKit?.
 - b) The encoded CloudKit metadata.
5. What's a good place to store server-change tokens?
 - c) UserDefaults.
6. Why is it smart to import CloudKit data on a background managed object context?
 - a) Because you don't know how long the import will take and you don't want to block the main thread.

Chapter 16

1. What is SceneKit typically used for?
 - c) 3D games.
2. What is a view in a SpriteKit game called?
 - b) A sprite.
3. What unit of measurement do SceneKit and ARKit use?
 - c) Centimeters.
4. Which tracking mode tracks both device movement and rotation?
 - c) World Tracking.
5. What should you keep in mind when you prepare images for image tracking in ARKit?
 - c) The images should have sufficient features.
6. What does ARKit need to accurately track objects?
 - c) Surfaces with lots of textures.

Chapter 17

1. What different types of location access exist?
 - a) Always and When in use.
2. When should you ideally ask for access to location services?
 - c) Once the user understands why you need access to their location.
3. How do you obtain a user's current location?
 - b) By calling `startUpdatingLocation()` on the location manager.

4. Which of the following location-tracking techniques is the most battery-friendly?
 - c) Visit tracking.
5. What different types of regions can you monitor?
 - a) Geofences and iBeacons.
6. Up to how long could it take for region-monitoring to start?
 - b) 10 minutes.
7. How can you check whether your app launched due to a significant location change?
 - b) By inspecting the app's launch options.

Chapter 18

1. What is the file format for a CoreML-compatible model?
 - b) .mlmodel.
2. How do you use a CoreML model in your project?
 - a) By adding the model to Xcode and using the model class that is generated for you.
3. Why is it so important the CoreML runs on the end user's device?
 - a) It's better for privacy and preserves the user's data plan.
4. What does the NaturalLanguage framework do?
 - b) It extracts interesting information from text, for instance, names, places, and words.
5. What does the Vision framework do?
 - c) It performs complex analyses on images.

6. Which of the following options could negatively impact the quality of a training set for an image classifier?
 - b) If the training subject is always shown in similar surroundings.

Chapter 19

1. Which devices support HealthKit?
 - c) The iPhone and Apple Watch.
2. What's the difference between a quantity sample and a category sample?
 - a) Quantity samples have an arbitrary value, while categories use enums to represent their value.
3. How can you check whether you have read access to a certain health metric?
 - c) To protect the user's privacy, there is no way to be sure.
4. How can you filter data that is returned from a HealthKit query?
 - a) With predicates.
5. What permissions do you need to use a workout route builder?
 - c) You need permission for both the builder and the workouts.
6. What do you need to keep in mind when you associate location data with a workout route?
 - b) You should make sure the location data is very accurate.

Chapter 20

1. How does Siri understand a user's request?
 - b) Siri uses natural language processing.
2. How are app extensions launched?
 - c) App extensions are launched by the extensions framework.

3. How can you share data between an app and an extension?
 - c) Both of the above.
4. What are the three stages of handling a Siri intent?
 - a) Resolve, Confirm, Handle.
5. How do you make sure Siri does not render its default UI?
 - b) By conforming IntentViewController to the INUIHostedViewSiriProviding protocol, and setting displaysMessage to true.
6. How can you donate actions to Siri?
 - a) Through user activities and interaction objects.
7. How can a user add their own voice commands for Siri Shortcuts?
 - a) Through the settings app.

Chapter 21

1. Which framework is used to implement a simple video player?
 - c) AVKit.
2. Which object is used to play a local audio file?
 - b) AVAudioPlayer.
3. What are remote commands, and where do they come from?
 - c) Media playback-related commands that can come from any external source, including the lock screen, control center, or even the user's headphones.
4. Which object can be used to quickly take a picture?
 - a) UIImagePickerController.

5. Which of the following is not needed to record a video with an `AVCaptureSession` object?
 - b) Permission to access the user's photo library.
6. What information tends to get lost when you convert a `UIImage` instance to a `CGImage` instance?
 - a) The orientation.

Chapter 22

1. How are `UNUserNotificationCenter` and `NotificationCenter` different?
 - a) One is responsible for showing notifications to users, and the other is a mechanism to send notifications about events inside an app.
2. What kinds of notifications can you send to a user?
 - a) Push notifications and local notifications.
3. What are provisional notifications?
 - c) Notifications that are delivered to the user quietly, so they can preview the notifications your app will send.
4. How does iOS group notifications?
 - a) It uses the thread identifier from the notification content and the app that the notification belongs to.
5. What does a notification service extension do?
 - c) It allows you to replace a notification's contents before the user sees it.
6. What does a notification content extension do?
 - b) It allows you to create a custom interface for when a user 3D-touches a notification.

Chapter 23

1. Where can users find Today Extensions?
 - c) By Swiping right on the lock screen or Notification Center, on the leftmost screen on the Springboard, and 3D-touching an app.
2. At what height are widgets displayed by default?
 - a) It depends on the screen width and user settings.
3. How can you allow a user to expand your widget's height?
 - c) By setting the largest available display mode on the extension context to expanded.
4. What is the best place to update your widget contents?
 - b) In `widgetPerformUpdate(completionHandler:)`.
5. How does iOS know the best moment to reload a widget's contents?
 - c) With the completion handler that is passed to `widgetPerformUpdate(completionHandler:)`.

Chapter 24

1. What devices support the full drag-and-drop experience?
 - b) Only iPad.
2. What kind of object do you have to provide to start a drag session?
 - a) `UIDragItem`.
3. How can you check whether your app can handle a certain drop action?
 - a) By checking the type of object that is associated with the drop action.

4. How is the drag-and-drop experience limited on iPhone?
 - c) Drag and drop is limited to the currently-open app.
5. What do you have to do to support drag and drop on an iPhone?
 - a) You must manually enable the drag-and-drop interactions.
6. How are drag and drop delegate protocols for collection views different from the regular drag and drop delegate protocols?
 - b) The collection view versions of the delegate methods receive an index path.

Chapter 25

1. Which objects are used to represent an item that you would like to index in Spotlight?
 - b) `NSUserActivity` and `CSSearchableItem`.
2. What happens when a user selects a Spotlight search result that points to your app?
 - a) `AppDelegate` is asked to resume a user activity that represents the item.
3. How do you add an item to the Spotlight index?
 - a) By passing `CSSearchableItem` to `CSSearchableIndex` or setting a user activity as the current activity.
4. Does it make sense to index the same user activity multiple times?
 - b) Yes, the more often an activity is performed in the app, the higher it will rank in the Spotlight search results.
5. What are Smart App Banners?
 - a) You can add Smart App Banners to web pages to point visitors to your app.

6. What are Universal Links?
 - a) Universal links are shareable web links that open your app if it's installed.
7. What happens when your app is expected to open a Universal Link?
 - b) AppDelegate is asked to open the URL.

Chapter 26

1. In what way are Message Extensions similar to apps?
 - a) They can exist independently and support In-App Purchases.
2. What do you need to create a Sticker Pack?
 - a) You need images that can be added to the Sticker Pack template.
3. How can you test your Message extensions?
 - b) You must use the predefined conversations in the simulator.
3. What object is used to configure a complex message that can be sent through your extension?
 - c) The MSMessages object.
4. How can you identify the users that participate in a conversation?
 - a) By reading their unique, random identifiers.
5. How do you know whether a message was delivered to the recipients?
 - c) There is no way to be sure due to privacy concerns.

Chapter 27

1. What does the Time Profiler do?
 - c) Shows you how much time is spent in certain methods.
2. What does the Allocations instrument do?
 - a) Shows you your app's memory usage over time.
3. Which of the following does not necessarily indicate a memory leak?
 - a) Growing memory in the memory graph.
4. What does Xcode's memory debugger show you?
 - b) The relationship between different objects that are in memory.
5. What are signpost logging?
 - a) A low-overhead log to measure your app's performance.
6. Which elements does an Instruments Package have?
 - a) Metadata, a schema, and the instrument itself.

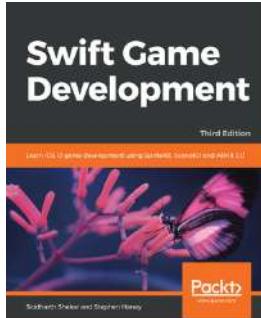
Chapter 28

1. What are the downsides of running certain code on the main thread?
 - a) It can block the UI.
2. In which circumstances will threading not help your performance?
 - a) When you have a lengthy layout-related calculation.
3. Which GCD feature can you use to execute an action when several other actions are done executing?
 - b) DispatchGroup.

4. Which of the following is not a benefit of operations?
 - b) Faster code execution.
5. What's the maximum number of Operations you can run at the same time?
 - a) There is no hard limit, iOS decides on the fly.
6. What is KVO?
 - a) A way for objects to observe properties of another object.
7. How do you start executing operations on an operation queue?
 - c) By adding operations to it.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Swift Game Development - Third Edition

Siddharth Shekar

ISBN: 9781788471152

- Deliver powerful graphics, physics, and sound in your game by using SpriteKit and SceneKit
- Set up a scene using the new capabilities of the scene editor and custom classes
- Maximize gameplay with little-known tips and strategies for fun, repeatable action
- Make use of animations, graphics, and particles to polish your game
- Understand the current mobile monetization landscape
- Integrate your game with Game Center
- Develop 2D and 3D Augmented Reality games using Apple's new ARKit framework
- Publish your game to the App Store



Hands-On Full-Stack Development with Swift

Ankur Patel

ISBN: 9781788625241

- Get accustomed to server-side programming as well as the Vapor framework
- Learn how to build a RESTful API
- Make network requests from your app and handle error states when a network request fails
- Deploy your app to Heroku using the CLI command
- Write a test for the Vapor backend
- Create a tvOS version of your shopping list app and explore code-sharing with an iOS platform
- Add registration and authentication so that users can have their own shopping lists

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

actions
 adding, to notifications 542, 545
app contents
 adding, to Spotlight index 598
App Store Connect
 application, adding 698, 699, 700, 701, 702, 703
App Transport Security (ATS) 323, 324
app
 indexing, through user activity 599, 601, 602
 packaging 703
 preparing, for launch 707, 708
 uploading, for beta testing 704, 705, 706
Apple Push Notification Service (APNS) 523
ARKit app
 content, placing in 3D space 405, 407, 409
 image tracking, adding 399
 image-tracking experience, building 401, 402, 403, 404
 images, preparing for tracking 400, 401
ARKit Quicklook view controller
 implementing 386, 387, 389, 390
ARKit Quicklook
 about 384
 using 386
ARKit
 about 382, 383
 content, rendering 383
 physical environment, tracking 384, 385
array literal
 set, creating 109
arrays
 about 77, 78
 creating, with initial values 79
 empty array, checking 86, 87

empty array, creating 78, 79
items, adding 80, 81, 82, 83
items, removing 92, 93, 94
iterating over 91, 92
mutable array, creating 79
number of elements, checking 85
value, retrieving 87, 88, 89, 90, 91
asynchronous code
 writing 681
audio controls
 implementing 498, 499, 500
audio player
 creating 497
 media, playing in background 504, 505, 506
 song metadata, displaying 502, 503
 time scrubber, implementing 501
audio
 playing 495
augmented reality 382
augmented reality apps
 features 383
Augmented Reality gallery
 implementing 399
auto layout 127

B

background fetch
 broad understanding 331
 prerequisites, implementing 331
 viewing, from distance 329, 330
 working 329
booleans (bools) 34

C

calendar-based notification
 scheduling 537, 538
cell

setting up 187, 188, 189, 206, 207, 208, 209, 210, 211
classes 251, 253, 254, 255, 256, 258
closed range 64, 65
CloudKit container 358
CloudKit dashboard
 exploring 354, 355
CloudKit data
 importing 374, 376
CloudKit database 355, 356, 357
CloudKit
 about 351
 adding, to MustC app 352, 353
 AppDelegate, configuring 366, 367
 changes, retrieving from 362, 364, 366
 combining, with Core Data 370
 communicating with 358, 359, 360
 Core Data models, preparing for 370, 372
 Core Data models, sending to 376, 379
 data, retrieving 358
 data, storing 358, 368, 369
 database changes 361
Collection View Controller
 adding 175, 176, 177
Collection View Controllers 123, 124
comments 40
comparison operators
 using 46
constant
 about 35
 versus variable 40
content extension
 adding, to app 553, 555, 557, 559
Continuity 598
control flow
 about 66
 for...in loop 67, 68, 69
 one-sided range 69, 71
 repeat...while loop 73, 74
 while loop 71, 72
controllers
 about 258
Collection View cells 269, 270, 271, 272
Collection View controllers 269, 270, 271, 272
 about 258
Collection Views 259, 260, 261, 262, 263, 265, 266
 creating 266, 267, 268
 data source 272, 273, 274
 data, displaying in Collection View 272
conversations 643
Core Data database
 data persistence 288
 data, filtering with predicates 295, 296
 data, reading 288
 data, reading with simple fetch request 292, 293, 295
 data, writing 288
 models, persisting 289, 290
 NSFetchedResultsController, implementing 297
 persistence code, refactoring 291, 292
 reacting, to database changes 296
Core Data model
 creating 282, 283, 284
 entities, using 286, 287
 preparing, for CloudKit 370, 372
 relationships, defining 285, 286
 sending, to CloudKit 376, 379
Core Data objects
 fetch logic, implementing 316, 317, 318
 movie, updating with popularity rating 320, 321
 multiple threads, visualizing 321, 322
 updating, with fetched data 314, 315
Core Data
 about 276
 adding, to application 279, 280, 281, 282
 stack 277, 278
Core Image
 photos, manipulating 514, 515, 516, 517, 518
Core Location framework 412
CoreML models
 obtaining 429
 using 429, 430, 432
CoreML
 about 426, 427, 428
 combining, with computer vision 432
CreateML
 models, training 437

natural language model, training 437, 438
Vision model, training 440, 441
critical notifications 522
CSSearchableItem instances
 adding, to search index 608, 610, 611
CSSearchableItem
 indexing with 605
CSSearchableItemAttributeSet
 information, containing 605, 606
custom color
 creating 183, 184, 185, 186
custom sticker app
 creating 630, 631, 632, 633
custom summary message
 providing, for notification group 547, 549
custom UI
 adding, to Siri 476, 477, 478

D

data types, Swift
 about 32
 booleans (bools) 34
 comments 40
 constant 35
 floating-point numbers 34
 integers (Ints) 33
 string 32, 33
 string, concatenating 41, 42
 string, interpolation 42, 43
 variable 35
data
 fetching, from web 307, 308
delivered notifications
 managing 541
dictionary
 about 94
 creating 95, 96
 dictionary keys and values, iterating 103
 dictionary keys, iterating over 103
elements, adding 97, 98, 100
elements, updating 97, 98, 100
item, accessing 100
items, removing 106, 107
number of items, checking 104
values, iterating over 101

dispatch queues
 using, in application 683, 684, 685, 687
drag and drop experience
 about 578
 customizing 592, 593
drag and drop
 adding, to plain UIView 583, 584, 586
 adding, to UICollectionView 586, 588, 589, 590, 592
 implementing 582

E

empty array
 checking 86, 87
 creating 78, 79
empty set
 creating 108
Explore cell
 Auto Layout, adding 236, 237
 setting up 235, 236
Explore header
 Auto Layout, adding 231, 233, 234
 setting up 222, 223, 224, 225, 227, 228, 229, 231

F

files
 creating, from scratch 138, 139
floating-point numbers 34
for...in loop 67, 68, 69
functions
 about 56
 creating 56, 57, 58, 59, 60, 61

G

geofences
 setting up 421, 422
grid
 updating 192, 193
grouped notifications
 about 524
 implementing 545

H

half-closed range 65, 66
HealthKit Store
access, requesting to 446, 447
HealthKit
about 445
data, retrieving 448, 449
data, storing 448, 449

I

if statement
about 48
using 48, 49, 50, 51, 52
image classifier
implementing 434, 435, 436
ImageAnalyzer 434
iMessage apps
about 627
custom compact view, implementing 635, 636, 637, 638
expanded view, implementing 639, 640, 641
implementing 634
interactive message 627
life cycle 634, 635
media content 627
message types 627
stickers 627, 628
iMessage sticker pack
creating 628, 629
indexing delegate 616
indexing methods
combining, safely 611, 612
inheritance 255
Instrument
creating 667
signpost logging, adding to app 668, 669, 670
Instruments Package
building 672
creating 673, 674, 675, 677
Instruments suite
exploring 650, 651, 653
integers (Ints)
about 33

comparison operators 46
decrementing 45
incrementing 45
operations, performing 43, 44, 45
Integrated Development Environment (IDE) 8
intents 456
intents extension
app extensions 459, 460
configuring 461, 462, 463
implementing 458
intents, handling in extension
about 466
desired action, performing 473, 475
intent status, combining 471, 472
user's input, resolving 467, 468, 469
interactions
Siri Shortcuts, donating 485, 487, 488, 489, 490
interface, Xcode
about 14
debug panel 15
device, connecting wirelessly 23, 24, 25
Generic iOS Device 18
iOS device, connecting 18, 19, 20, 21, 22
navigator panel 15
standard editor 15
toolbar 16
utilities panel 15
window pane controls 25, 26, 27
iOS
Today Extension, finding 564

J

JSON
in Swift 311, 312, 313

K

KVO (Key-Value Observing) 690

L

Let's Eat app
building 128
Explore tab 128, 129
location, selecting 130
Map tab 133

project, creating 134, 135, 136
project, setting up 134
restaurant detail, obtaining 132
restaurants, listing 130, 131

local notifications

about 523
creating 533, 534

location changes

subscribing to 418, 419, 420

location-based notification

scheduling 538

Locations cell

setting up 247

M

machine learning 426, 427

machine learning, Apple
reference 428

managed object context 278

Map Kit View

setting up 216, 217, 219, 220

memory leaks

about 658
avoiding 663, 664, 665, 666
closing 658
discovering 662

messages

about 643
composing 644
sending 645, 646

modal

adding 194, 195, 196, 197, 198, 199, 200

Bar Button Items, updating 200

Cancel button, unwinding 201, 202

Table View, adding 202, 203

Model View Controller (MVC)

about 127, 249
architecture 249, 250, 251
Controller 250
Model 250
setting up 251
View 250

models

training, with CreateML 437

Mojave 437

Mosaic app 658

movie cell

rating, adding to 323

movie ratings

changes, observing to 324

MustC app

background fetch capability, adding 332, 333

CloudKit, adding to 352, 353

data model, updating 335, 337

existing code, refactoring 338

helper struct, preparing 340

iOS, used for waking up 333

movies, updating 344, 345

movies, updating in background 334, 340

mutable array

creating 79

mutable set

creating 109

N

natural language model

training 437, 438

Navigation Controller

124

neural network

427

notification contents

creating 532

notification extensions

implementing 550

notification group

custom summary message, providing for

547, 549

notifications

about 522

actions, adding to 542, 545

calendar-based notification 537

forms 522

grouped notifications 545

grouping, based on thread identifiers 546

handling 526, 539

handling, in app 539, 541

local notifications 533

location-based notification 538

push notifications 532

registering for 526, 527, 528, 529, 531, 532

scheduling 526, 534

timed notification 535
NSFetchedResultsController
implementing 297, 299, 300
NSManagedObjectContext
multiple instances 303, 304
NSUserActivity
Siri Shortcuts, implementing through 480,
482, 483, 484

O

objects
preventing, from using infinite memory 658,
659
one-sided range 69, 71
Operations
reusable tasks, creating 688
using, in apps 689, 690, 692, 693
optional binding 53, 54
optionals
need for 55
using 53, 54, 55

P

pending notifications
managing 541
persistent store 278
persistent store coordinator 278
photos
manipulating, with Core Image 514, 515,
516, 517, 518
pictures
storing 508
taking 507, 508
plain UIView
drag and drop, adding 583, 584, 586
Playground
about 29
interface 31, 32
project, creating 48, 63
using 29, 30, 31
predicate 295
protocol 272
provisional notifications 523
push notifications 523
creating 532, 533

R

ranges
about 63
closed range 64, 65
half-closed range 65, 66
reference cycles
avoiding 660
reference type 255
repeat...while loop 73, 74
Restaurant cell
Auto Layout, adding 243, 244, 245, 246
setting up 237, 238, 240, 242, 243
Restaurant List View
adding 205
reusable tasks
creating, with Operations 688
Reviews View
adding 212, 213
reviews
viewing 214, 215

S

SceneKit scene
creating 396, 397, 398
SceneKit
exploring 395
Search Engine Optimization (SEO) 597
searchable item selection
handling 612
section header
about 189
using 189, 190, 191, 192
segues 127
server
preparing, for Universal Links 618, 619, 620
service extension
adding, to app 550, 551, 552
sessions 643
sets
about 108
creating, with array literal 109
empty set, creating 108
intersecting 116, 117
items, adding 110, 111, 112

items, checking 112, 113
items, removing 119, 120
iterating over 113, 114, 115
joining 117, 118
mutable set, creating 109

Siri Shortcuts
donating, with interactions 485, 487, 488, 489, 490, 491
implementing 479
implementing, through NSUserActivity 480, 482, 483, 484

Siri
about 456, 457
custom UI, adding to 476, 477, 479
domains 456
new vocabularies, teaching at runtime 465

slow code
discovering 654, 655, 656, 657

Spotlight index
app contents, adding to 597, 598

Spotlight search 595, 596, 597

Spotlight
best practices 614
indexing delegate 616
metadata, adding to web links 614, 615
ratings 614

sprite 392

SpriteKit scene
creating 394, 395

SpriteKit
exploring 392, 393, 394

Stack Views 127

stickers
assets, optimizing for 629, 630

Storyboard
about 126
app assets, adding 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157

ExploreViewController file, linking 177, 178, 180, 181, 182, 183

Launch screen, creating 158, 159, 160, 161, 162, 163, 164, 165, 166

Navigation Controller, adding 167, 168, 169, 170

RestaurantViewController file, linking 205, 206

setting up 140, 141, 142, 143, 144

UIViewController file, linking 177, 178, 180, 181, 182, 183

updating 157

string
about 32, 33
concatenating 41, 42
interpolation 42, 43

structures (structs) 251, 253, 254, 255, 256, 257, 258

support vector machine 427

Swift types 253

Swift
data types 32

T

Tab Bar Controllers 125

Table View Controller 122

threads 681, 682, 683

timed notification
scheduling 535, 536

Today Extension
about 566
adding, to app 567, 568, 570, 572, 573
anatomy 562, 564
finding, in iOS 564

Trekker app
implementing 450

type inference 41

type safety 41

U

UICollectionView
drag and drop, adding 586, 588, 589, 590, 592

UIDragInteractionDelegate 579, 580

UIDropInteractionDelegate 580, 581, 582

Universal Links
handling, in app 620, 621, 622
server, preparing for 618, 619, 620
visibility, increasing of app 617, 618

URLSession
basics 308, 309, 310

user activity

app, indexing through 598, 599, 600, 602
user location
obtaining 416, 417
permission, asking to access location data
413, 414, 415
requesting 413

V

value type 255
variable
about 35
Boolean, creating 38, 39
creating, with integer (int) 35
creating, with string 35
debugging 36
floating-point numbers, adding 36, 37
print() 36
versus constant 40
video player
creating 495, 496, 497
video
playing 495
recording 507, 510, 511, 512, 513, 514

storing 510, 511, 513, 514
View Controllers 122
Vision framework 433, 434
Vision model
training 440, 441
vocabularies
adding, through .plist file 463, 464
adding, to app 463

W

web
data, fetching from 307, 308
while loop 71, 72
workout app
implementing 450, 452
world tracking 385

X

Xcode
about 8
downloading 9, 10, 11, 12, 13, 14
interface 14